МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **ПОРІВНЯЛЬНИЙ АНАЛІЗ PWA-ЗАСТОСУНКІВ З ТРАДИЦІЙНИМИ ВЕБ-ЗАСТОСУНКАМИ**

Виконав: студент 2 курсу, групи 8.1214-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
О. Ю. Яндола
(ініціали та прізвище)
Керівник доцент А.І. Безверхий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра __електроніки, інформаційних систем та програмного__ забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри _____Т.В. Критська_
"_01_" __квітня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Яндолі Олександру Юрійовичу
(прізвище, ім'я, по батькові)

1.Тема роботи _Порівняльний аналіз PWA-застосунків з традиційними веб-застосунками

керівник роботи _____доцент Безверхий Анатолій Ігорович____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від _28.03. 2025 р. № 361-е__
2. Строк подання студентом кваліфікаційної роботи_____01.12.2025 р.___
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- аналіз літератури та підходів до порівняння PWA-застосунків з традиційними веб-застосунками
- огляд рішень у сфері PWA застосунків
- дослідження організації бізнес-логіки веб-застосунків;
- розробка програмних продуктів для порівняння PWA-застосунків з традиційними веб-застосунками
- визначення вимог до системи
- тестування, аналіз результатів і формування рекомендацій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.04.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Вибір теми та формулювання цілей дипломної роботи	15.04 – 20.04.25	виконано
2	Аналіз літератури та сучасних підходів до PWA-застосунків та традиційних веб-застосунків.	21.04 – 30.04.25	виконано
3	Огляд технологій і інструментів мікросервісної архітектури.	01.05 – 10.05.25	виконано
4	Проєктування PWA-застосунку.	11.05 – 20.05.25	виконано
5	Реалізація PWA-застосунків.	21.05 – 06.08.25	виконано
6	Реалізація традиційних веб-застосунку.	01.09 – 30.09.25	виконано
7	Тестування продуктивності та масштабованості	01.10.25	виконано
8	.Порівняльний аналіз PWA-застосунку з традиційними веб-застосунками	02.10 – 31.10.25	виконано
9	Виявлення та усунення проблем інтеграції та безпеки.	01.11 – 25.11.25	виконано
10	Оформлення пояснювальної записки.	01.12 – 12.12.25	виконано

Студент _____ Яндола О.Ю.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Безверхий А.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А,__
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 108

Рисунків: 24

Таблиць: 14

Лістингів коду: 25

Джерел: 50

Яндола О. Ю. Порівняльний аналіз PWA-застосунків з традиційними веб-застосунками: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник А. І. Безверхий. Запоріжжя : ЗНУ, 2025. 108 с.

Метою дослідження є розробка PWA-додатку «City Explorer» для дослідження міських подій, місць відпочинку та погоди з інтеграцією сучасних API та аналізом продуктивності PWA порівняно зі звичайними веб-додатками. Запропонований підхід до реалізації офлайн-функціональності та оптимізації швидкодії демонструє ефективність використання PWA-технологій для створення доступних і продуктивних рішень у сфері міського туризму. Практичне значення роботи полягає у створенні функціонального PWA-додатку «City Explorer», який може бути використаний для планування міських подій і дозвілля. Додаток забезпечує доступ до актуальної інформації про погоду, місця відпочинку та події, а також працює в офлайн-режимі, що підвищує його зручність для користувачів. Результати порівняльного аналізу продуктивності PWA можуть бути використані розробниками для обґрунтування вибору технологій при створенні веб-додатків.

Ключові слова: PWA, веб-застосунки, аналізом продуктивності, Node.js, React, API.

SUMMARY

Pages: 108

Figures: 24

Tables: 14

Code Listings: 25

References: 50

Yandola O. Yu. Comparative Analysis of PWA Applications with Traditional Web Applications: Master's thesis in specialty 121 "Software Engineering" / Scientific supervisor A. I. Bezverkhyi. Zaporizhzhia: ZNU, 2025. 108 p.

The aim of the study is to develop a PWA application "City Explorer" for exploring urban events, recreational places, and weather with integration of modern APIs and analysis of PWA performance compared to conventional web applications. The proposed approach to implementing offline functionality and optimizing performance demonstrates the effectiveness of using PWA technologies for creating accessible and productive solutions in the field of urban tourism. The practical significance of the work lies in creating a functional PWA application "City Explorer," which can be used for planning urban events and leisure activities. The application provides access to up-to-date information about weather, recreational places, and events, and also operates in offline mode, which enhances its convenience for users. The results of the comparative performance analysis of PWA can be used by developers to justify the choice of technologies when creating web applications.

Keywords: PWA, web applications, performance analysis, Node.js, React, API.

ЗМІСТ

ВСТУП 10
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАНЬ 15
1.1 Актуальність розробки PWA-додатків у сучасних умовах 15
1.2 Огляд існуючих веб- та мобільних додатків для пошуку подій, місць і погоди 17
1.2.1 Додатки для пошуку погоди 17
1.2.2 Додатки для пошуку місць (POI) 19
1.2.3 Додатки для пошуку подій 21
1.2.4 Інтегровані додатки 22
1.3 Переваги та обмеження Progressive Web Apps порівняно з нативними додатками 23
1.3.1 Переваги PWA порівняно з нативними додатками 24
1.3.2 Обмеження PWA порівняно з нативними додатками 25
1.3.3 Переваги нативних додатків 25
1.3.4 Порівняльна таблиця 25
1.4 Аналіз сучасних геоінформаційних API (Geoapify, WeatherAPI, Ticketmaster Discovery API) 26
1.4.1 Аналіз Geoapify API 26
1.4.2 Аналіз WeatherAPI 27
1.4.3 Аналіз Ticketmaster Discovery API 28
1.4.4 Порівняльна таблиця функцій API 28
1.5 Вимоги до офлайн-функціональності та кешування в PWA 29
1.5.1 Основні вимоги до офлайн-функціональності в PWA 29
1.5.2 Стратегії кешування в PWA 30
1.5.3 Вимоги до персистенції даних (Redux Persist з IndexedDB) 31
1.6 Виклики та рекомендації 32
РОЗДІЛ 2. ВИБІР ТЕХНОЛОГІЙ ТА ПРОЕКТУВАННЯ PWA-ДОДАТКУ 33
2.1 Обґрунтування вибору React, Redux Toolkit, Vite для клієнтської частини 33

2.2 Обґрунтування використання Node.js, Express, MongoDB для серверної частини 36
2.3 Порівняння інструментів створення PWA (Vite + Workbox, CRA, Next.js) 38
2.4 Проектування архітектури додатку (клієнт-сервер, Redux slices, API-шлюзи) 40
2.5 Проектування схеми бази даних та системи автентифікації 43
2.6 Проектування механізмів офлайн-кешування (Service Worker, IndexedDB, localForage) 44
РОЗДІЛ 3 РОЗРОБКА PWA-ДОДАТКУ З ВИКОРИСТАННЯМ REACT І NODE.JS 47
3.1 Розробка серверної частини (реєстрація, авторизація, favorites, API-проксі) 47
3.2 Налаштування PWA (Vite Plugin PWA, manifest.json, Service Worker, Workbox) 58
3.3 Розробка клієнтської частини: інтерфейс, карта (Leaflet), пошук, погода, події 65
3.4 Інтеграція з Geoapify (autocomplete, places, routing, my-location) 78
3.5 Інтеграція з WeatherAPI та Ticketmaster Discovery API 84
3.6 Реалізація геолокації, маршрутів, обраного та офлайн-режиму 87
РОЗДІЛ 4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО PWA-ДОДАТКУ 93
4.1 Методика проведення досліджень (Lighthouse, Chrome DevTools, реальні пристрої) 93
4.2 Дослідження продуктивності в онлайн- та офлайн-режимах 97
4.3 Порівняння метрик роботи з аналогами (Google Maps) 99
4.4 Аналіз економічної доцільності використання PWA замість нативних додатків 101
ВИСНОВКИ 102
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 104

ВСТУП

Актуальність теми

У сучасному цифровому світі веб-додатки стали невід'ємною частиною повсякденного життя, забезпечуючи доступ до інформації, сервісів і розваг у будь-який час і з будь-якого пристрою. Зростання популярності мобільних пристроїв і підвищення вимог користувачів до швидкості, доступності та зручності веб-додатків спричинили потребу в нових технологічних рішеннях, здатних конкурувати з нативними додатками. Progressive Web Apps (PWA) є відповіддю на ці виклики, поєднуючи переваги веб-технологій, такі як кросплатформність і легкість розгортання, з можливостями нативних додатків, включаючи офлайн-доступ, push-повідомлення та високу продуктивність. PWA дозволяють створювати додатки, які завантажуються миттєво, працюють у режимі офлайн і забезпечують безперебійний користувацький досвід навіть за нестабільного інтернет-з'єднання.

Розробка PWA-додатків із використанням сучасних технологій, таких як React для створення інтерактивних інтерфейсів, Vite для швидкого збирання та оптимізації фронтенду, і Node.js для побудови ефективних серверних рішень, відкриває нові перспективи для створення масштабних і функціональних систем. Інтеграція сучасних API, таких як Weather API, Geoapify і Ticketmaster, дозволяє значно розширити функціональність додатків, надаючи користувачам доступ до актуальної інформації в реальному часі. Особливої актуальності набуває сфера міського туризму та планування дозвілля, де користувачі потребують швидкого доступу до даних про події, місця відпочинку та погодні умови. Крім того, аналіз продуктивності PWA у порівнянні з традиційними веб-додатками є важливим для оцінки їхньої ефективності та обґрунтування вибору технологій у процесі розробки. Дослідження в цій області не лише сприяє вдосконаленню користувацького досвіду, але й має потенціал для трансформації підходів до створення веб-додатків у різних галузях, включаючи туризм, освіту та комерцію. Таким

чином, актуальність теми дослідження зумовлена необхідністю створення високопродуктивних, доступних і функціональних веб-додатків, які відповідають сучасним технологічним трендам і очікуванням користувачів.

Мета і завдання дослідження

Метою дослідження є розробка PWA-додатку «City Explorer» для дослідження міських подій, місць відпочинку та погоди з інтеграцією сучасних API та аналізом продуктивності PWA порівняно зі звичайними веб-додатками.

Для досягнення поставленої мети визначено наступні завдання:

1. Розробити клієнтську частину додатку з використанням React і Vite для створення інтерактивного інтерфейсу та оптимізації процесу розробки й збирання.
2. Побудувати серверну частину на базі Node.js та Express.js для обробки запитів.
3. Інтегрувати сучасні API (Weather API, Geoapify, Ticketmaster) для отримання актуальних даних.
4. Реалізувати PWA-функціональність, включаючи офлайн-режим і кешування.
5. Провести порівняльний аналіз продуктивності PWA-додатку зі звичайним веб-додатком за показниками швидкості завантаження, доступності в офлайн-режимі, споживання ресурсів тощо.

Об'єкт дослідження

Об'єктом дослідження є Progressive Web Apps (PWA) як технологія створення веб-додатків із розширеними функціональними можливостями.

Предмет дослідження

Предметом дослідження є процес розробки PWA-додатку з використанням React і Node.js, інтеграція сучасних API та порівняння продуктивності PWA зі звичайними веб-додатками.

Методи дослідження

Для вирішення поставлених завдань використано наступні методи:

1. Аналіз літератури та документації для вивчення сучасних технологій розробки PWA, зокрема React, Node.js, Service Workers і Workbox.
2. Синтез отриманих знань для проектування архітектури додатку «City Explorer».
3. Експериментальне програмування для реалізації клієнтської та серверної частин додатку.
4. Інтеграція API для забезпечення функціональності додатку.
5. Порівняльний аналіз продуктивності PWA та традиційного веб-додатку за допомогою інструментів тестування (Lighthouse, WebPageTest).

Наукова новизна одержаних результатів

Наукова новизна дослідження полягає в розробці PWA-додатку «City Explorer» з інтеграцією сучасних API та проведенні порівняльного аналізу продуктивності PWA зі звичайними веб-додатками. Запропонований підхід до реалізації офлайн-функціональності та оптимізації швидкодії демонструє ефективність використання PWA-технологій для створення доступних і продуктивних рішень у сфері міського туризму.

Практичне значення одержаних результатів

Практичне значення роботи полягає у створенні функціонального PWA-додатку «City Explorer», який може бути використаний для планування

міських подій і дозвілля. Додаток забезпечує доступ до актуальної інформації про погоду, місця відпочинку та події, а також працює в офлайн-режимі, що підвищує його зручність для користувачів. Результати порівняльного аналізу продуктивності PWA можуть бути використані розробниками для обґрунтування вибору технологій при створенні веб-додатків.

Апробація одержаних результатів

Результати дослідження були представлені на науково-практичній конференції студентів, аспірантів і молодих учених Запорізького національного університету «Молода наука-2025», а також на науково-технічній конференції Інженерного навчально-наукового інституту ЗНУ.

Глосарій

API (Application Programming Interface) – програмний інтерфейс, який дозволяє різним додаткам або сервісам обмінюватися даними та виконувати певні функції.

Express.js – фреймворк для Node.js, який спрощує створення серверних додатків шляхом надання інструментів для обробки HTTP-запитів і маршрутизації.

Node.js – середовище виконання JavaScript, яке дозволяє створювати серверні додатки за межами браузера.

Progressive Web App (PWA) – веб-додаток, який використовує сучасні веб-технології для забезпечення функціональності, подібної до нативних додатків, включаючи офлайн-режим і push-повідомлення.

React – JavaScript-бібліотека для створення інтерактивних користувацьких інтерфейсів, заснована на компонентному підході.

Redux – бібліотека для управління станом додатку, яка забезпечує передбачувану поведінку даних у React-додатках.

Service Worker – скрипт, який працює у фоновому режимі браузера, забезпечуючи кешування ресурсів і офлайн-функціональність PWA.

Workbox – набір бібліотек для спрощення роботи з Service Workers, зокрема для кешування та управління стратегіями завантаження ресурсів.

manifest.json – файл конфігурації PWA, який визначає метадані додатку, такі як назва, іконки та налаштування відображення.

MongoDB – документоорієнтована база даних, яка використовується для зберігання даних у форматі JSON-подібних документів.

WeatherAPI – API для отримання даних про погоду в реальному часі, включаючи температуру, опади та прогноз.

Geoapify – API для пошуку та отримання інформації про місця, такі як ресторани, парки чи туристичні об’єкти.

Ticketmaster – API для доступу до інформації про події, включаючи дати, місця проведення та описи.

Кешування – процес зберігання даних у тимчасовій пам’яті для швидшого доступу при повторних запитах.

Офлайн-режим – функціональність PWA, яка дозволяє додатку працювати без підключення до мережі за рахунок попередньо кешованих даних.

Vite – сучасний інструмент для створення та збирання веб-додатків, який використовує ES-модулі для швидкого запуску серверу розробки та оптимізованого збирання проєктів.

Vite Plugin PWA – плагін для Vite, який спрощує налаштування PWA-функціональності, включаючи конфігурацію Service Workers і manifest.json для забезпечення офлайн-режиму та кешування.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАНЬ

1.1 Актуальність розробки PWA-додатків у сучасних умовах

У сучасному цифровому ландшафті, де мобільні пристрої домінують у повсякденному житті користувачів, розробка Progressive Web Apps (PWA) набуває особливого значення. За даними Statista, у другому кварталі 2025 року мобільні пристрої (без урахування планшетів) генерували 62,54% глобального веб-трафіку, що свідчить про стійке зростання порівняно з 6,1% у 2011 році та 38,59% у 2015 році [1]. У липні 2025 року цей показник сягнув 64,35% [2], а в Сполучених Штатах Америки мобільний трафік становить 47,3%, хоча десктоп все ще лідирує з 50,2% [3]. Таке домінування мобільного веб-використання підкреслює необхідність створення додатків, які забезпечують швидкий, надійний і зручний доступ до інформації незалежно від якості інтернет-з'єднання.

Progressive Web Apps представляють собою гібридну технологію, що поєднує переваги веб-сайтів (кросплатформність, легкість розгортання та індексацію пошуковими системами) з функціональністю нативних мобільних додатків (офлайн-доступ, push-повідомлення, повноекранний режим і висока продуктивність). За прогнозами Straits Research, глобальний ринок PWA у 2024 році оцінювався в 3,53 млрд доларів США, а у 2025 році очікується зростання до 5,23 млрд доларів, з подальшим збільшенням до 21,44 млрд доларів до 2033 року [4]. Інші джерела, такі як Research Nester, оцінюють ринок PWA у 2025 році в 2,8 млрд доларів [5], тоді як загальний прогноз сягає понад 15 млрд доларів [6]. Це зростання обумовлене кількома факторами: покращенням підтримки браузерами (наприклад, Chrome, Safari та Edge), зниженням витрат на розробку порівняно з нативними додатками та підвищенням вимог користувачів до швидкості завантаження.

Актуальність PWA особливо помітна в умовах нестабільного інтернету, що є характерним для багатьох регіонів, включаючи Україну. За даними GSMA, мобільні технології та сервіси генерують близько 5,8% глобального ВВП, що становить 6,5 трлн доларів економічної доданої вартості у 2025 році [7]. PWA дозволяють зменшити залежність від постійного підключення, що критично для додатків, пов'язаних з геолокацією, пошуком подій та прогнозом погоди — як у випадку з розробленим у цій роботі додатком "City Explorer". Наприклад, успішні кейси, такі як Twitter Lite (тепер X Lite), демонструють зменшення використання даних на 70% та зростання залученості користувачів на 65% порівняно з нативним додатком [8]. Аналогічно, Forbes зазначає, що в Північній Америці у 2025 році лише 52,89% веб-трафіку припадає на мобільні пристрої, але глобальна тенденція до мобільності продовжує зростати [9].

Розробка PWA стає стратегічним вибором для бізнесу та розробників, оскільки дозволяє досягти ширшої аудиторії без необхідності публікації в App Store чи Google Play, що скорочує час на ринок і витрати. Згідно з опитуванням SlashData у першому кварталі 2025 року, мотивації розробників для використання PWA включають покращення користувацького досвіду, зниження витрат на підтримку кількох платформ та інтеграцію з сучасними API [10]. У контексті магістерського дослідження, актуальність полягає в демонстрації, як PWA можуть ефективно інтегрувати геоінформаційні сервіси (наприклад, Geoapify для пошуку місць і маршрутів, WeatherAPI для прогнозу погоди та Ticketmaster для подій), забезпечуючи офлайн-функціональність через Service Worker та IndexedDB, як реалізовано в проекті.

Однак, попри переваги, PWA стикаються з викликами, такими як обмежена підтримка певних функцій на iOS (наприклад, повноцінні push-повідомлення) та необхідність оптимізації для різних пристроїв. У 2025 році, з ростом мобільного трафіку до 64,35%, PWA стають не просто альтернативою, а стандартом для сучасних веб-додатків, що підкреслює їхню актуальність для досліджень у сфері веб-розробки.

1.2 Огляд існуючих веб- та мобільних додатків для пошуку подій, місць і погоди

У сучасному ринку мобільних і веб-додатків існує велика кількість рішень, призначених для пошуку локальних подій, місць (points of interest, POI) та прогнозу погоди. Ці додатки часто інтегрують геолокацію, API для реального часу та елементи соціальних мереж, щоб забезпечити користувачам зручний доступ до інформації. За даними Itransition, глобальний ринок мобільних додатків оцінюється в $289 млрд у 2025 році, з значною часткою у категоріях подорожей, навігації та подій [11]. Аналіз існуючих рішень дозволяє виділити ключові тенденції: перехід до PWA для офлайн-доступу, інтеграцію AI для персоналізації та фокус на локальному контенті. Нижче розглянуто основні категорії додатків, їх функціональність, переваги та обмеження, з акцентом на аналоги розробленого в роботі додатку "City Explorer".

1.2.1 Додатки для пошуку погоди

Додатки для прогнозу погоди є одними з найбільш популярних, з мільйонами активних користувачів. Наприклад, The Weather Channel має понад 100 млн активних щомісячних користувачів у 2025 році [12]. Вони часто включають локальні прогнози, радари та сповіщення про негоду.

- AccuWeather: Один з лідерів ринку, з акцентом на гіперлокальні прогнози (MinuteCast — прогноз на хвилини). Підтримує офлайн-режим для базових даних, інтеграцію з геолокацією та віджети. У 2025 році оновлений інтерфейс включає AI для персоналізованих сповіщень, з понад 50 новими функціями, включаючи покращений радар і ексклюзивні блискавкові сповіщення [13, 14]. Переваги: висока точність, підтримка PWA. Обмеження: реклама в безкоштовній версії, високе споживання батареї.

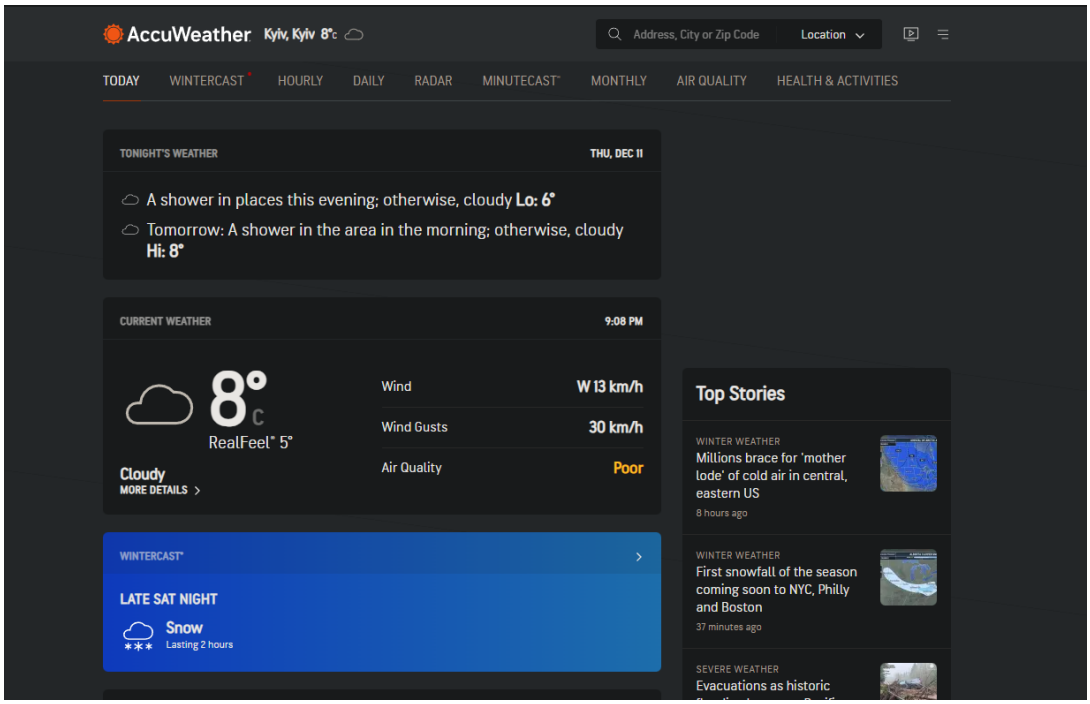


Рисунок 1 – Сторінка AccuWeather

- The Weather Channel: Інтегрує IBM Watson для аналізу даних. Пропонує радари, відео-прогнози та інтеграцію з подіями (наприклад, вплив погоди на заходи). У 2025 році додана функція "GoRun Index" для спорту. Переваги: детальні карти, PWA-підтримка. Обмеження: надмірна реклама, менша точність у віддалених регіонах[15].

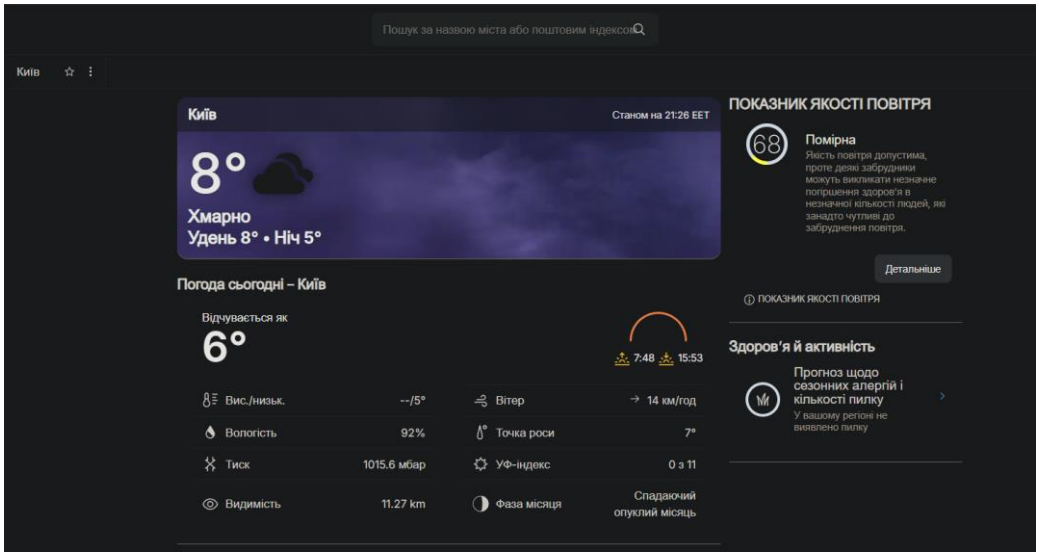


Рисунок 2 – Сторінка The Weather Channel

- Carrot Weather: Фокус на гумористичному інтерфейсі з AI-генерованими жартами. Підтримує Apple Watch, офлайн-кешування та інтеграцію з Dark Sky API. У 2025 році оновлено для AR-прогнозів. Переваги: креативність, точність. Обмеження: преміум-модель, обмежена для Android. [16]

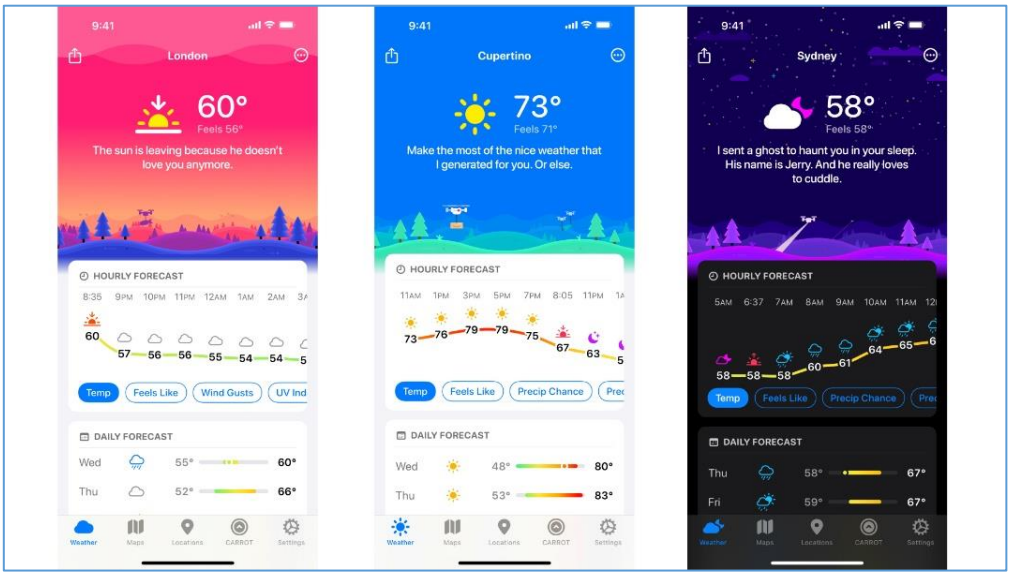


Рисунок *3* – Сторніка Carrot Weather

1.2.2 Додатки для пошуку місць (POI)

Додатки для пошуку місць часто поєднують навігацію, відгуки та рекомендації.

- Google Maps: Домінуючий гравець з інтеграцією місць, маршрутів, погоди та подій. Підтримує офлайн-карти, AR-навігацію (Live View) та PWA-версію (Google Maps Go). У 2025 році додана AI для персоналізованих маршрутів з Gemini, включаючи hands-free conversational navigation [17, 18]. Переваги: всебічна інтеграція, точність. Обмеження: залежність від Google-акаунту, проблеми з приватністю.

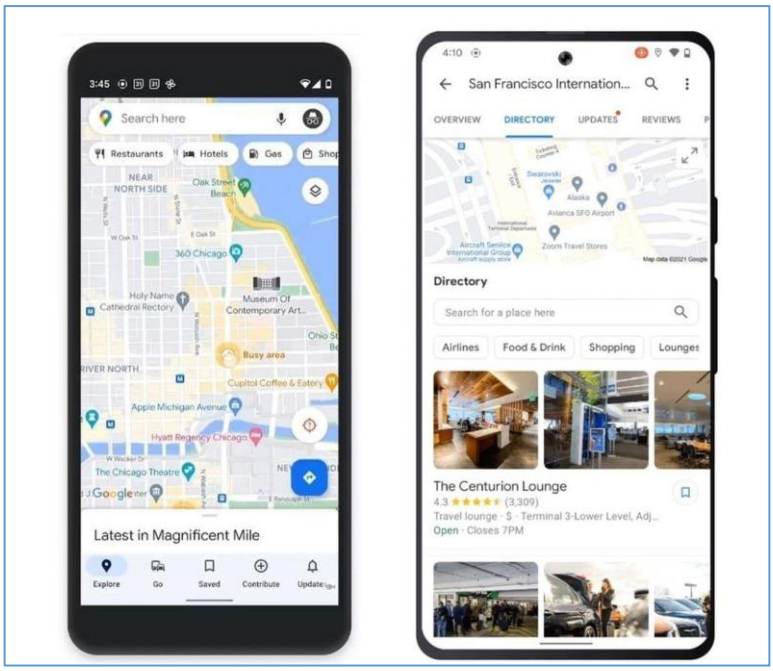


Рисунок *4* – Google Maps

- TripAdvisor: Фокус на відгуках про місця (готелі, ресторани, пам'ятки). Інтегрує події та бронювання. PWA-підтримка для веб-версії. У 2025 році AI-допомічник для планування поїздок, включаючи ChatGPT-інтеграцію та AI для виявлення фейкових відгуків [19]. Переваги: спільнота відгуків. Обмеження: фокус на туризмі, менша точна геолокація.

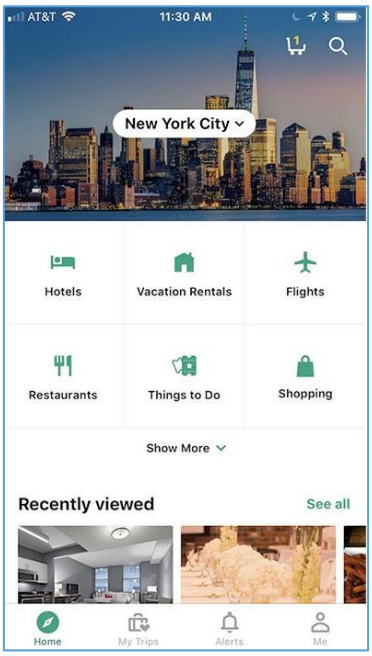


Рисунок *5* – TripAdvisor

- Yelp: Локальні пошуки ресторанів, магазинів з відгуками. Інтегрує події. PWA для веб. Переваги: соціальний аспект. Обмеження: реклама, обмежена поза США.[20]

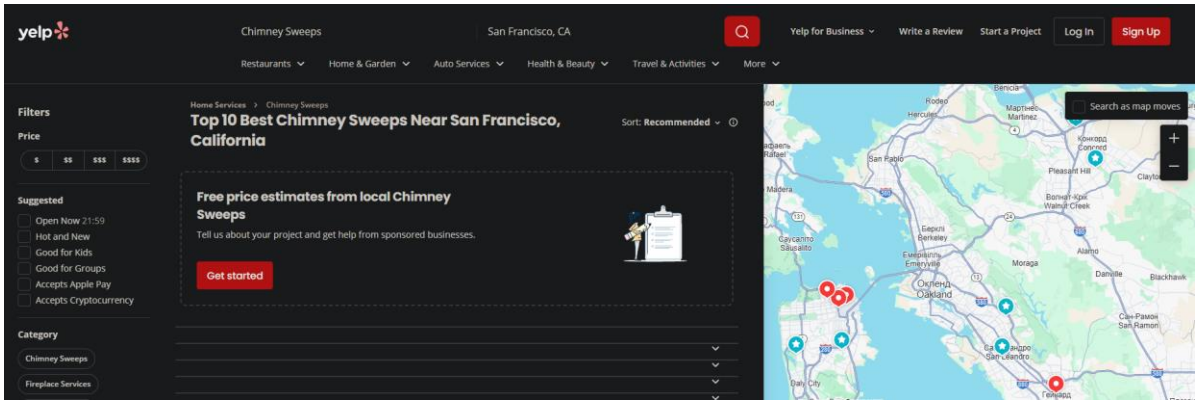


Рисунок *6* – Yelp

1.2.3 Додатки для пошуку подій

Ринок додатків для подій зріс, з event app market $1.86 млрд у 2025 році [21].

- Eventbrite: Лідер для пошуку та бронювання подій (концерти, фестивалі). Персоналізовані рекомендації, інтеграція з календарем. PWA-підтримка. У 2025 році оновлений інтерфейс з Lineup Tool для музики та партнерством з Listener.com для кращого відкриття [22, 23]. Переваги: великий каталог, соціальне поширення. Обмеження: комісії за квитки, менш фокус на малих подіях.

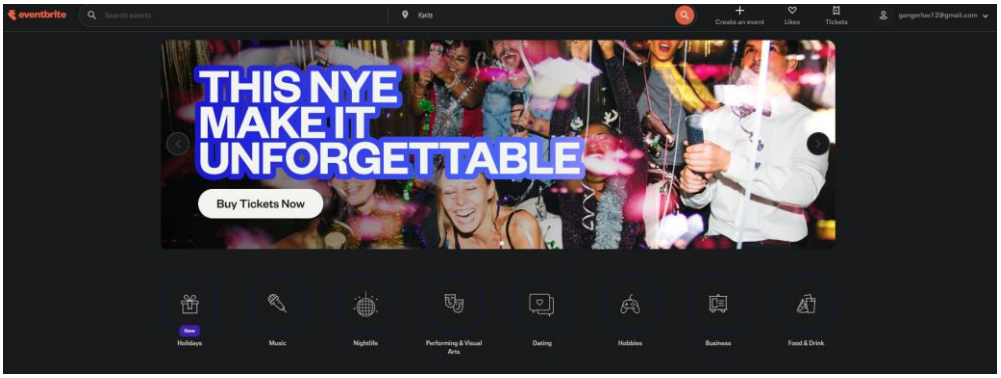


Рисунок *7* – Eventbrite

- AllEvents: Спеціалізується на локальних подіях, з фільтрами за геолокацією. Підтримує спільноту. У 2025 році додана функція Event Discussions для з'єднання з організаторами та оновлений homepage з динамічними привітаннями [24, 25]. Переваги: різноманітність, безкоштовний доступ. Обмеження: обмежена інтеграція з погодою.

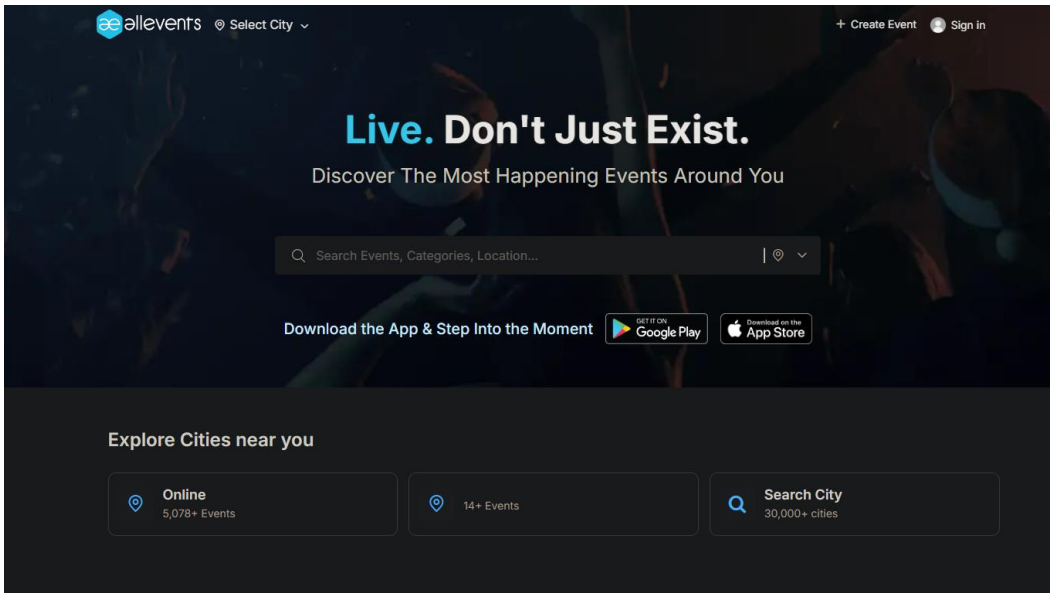


Рисунок *8* – AllEvents

1.2.4 Інтегровані додатки

Деякі додатки поєднують всі функції: Google Maps (місця + погода + події), TripAdvisor (місця + події). Вони часто мають PWA-версії для офлайн, але рідко повноцінно інтегрують всі аспекти з офлайн-кешуванням, як у "City Explorer" (див. Таблиця 1).

Таблиця 1 – Порівнялная таблиця

Додаток	**Функції (місця/події/погода)**	**PWA-підтримка**	**Офлайн-режим**	**Переваги**	**Обмеження**
Google Maps	Так/Частково/Так	Так	Так (карти)	Всебічна інтеграція	Приватність
AccuWeather	Ні/Ні/Так	Так	Частково	Точність прогнозів	Реклама

Eventbrite	Частково/Так/Ні	Так	Ні	Бронювання подій	Комісії
TripAdvisor	Так/Так/Ні	Так	Ні	Відгуки спільноти	Туристичний фокус
AllEvents	Частково/Так/Ні	Ні	Ні	Локальні події	Обмежена інтеграція

Ці додатки демонструють тенденцію до інтеграції, але мало хто пропонує повний офлайн-режим для всіх функцій, що підкреслює актуальність розробки PWA як "City Explorer".

1.3 Переваги та обмеження Progressive Web Apps порівняно з нативними додатками

Progressive Web Apps (PWA) та нативні мобільні додатки представляють два основні підходи до розробки мобільних рішень, кожен з яких має свої сильні та слабкі сторони. У 2025 році, з ростом мобільного трафіку до 64,35% глобально, вибір між PWA та нативними додатками залежить від вимог проекту, таких як продуктивність, вартість та охоплення аудиторії. PWA, побудовані на веб-технологіях (HTML, CSS, JavaScript), дозволяють створювати додатки, які працюють як веб-сайти, але з можливостями нативних (офлайн, push-повідомлення, додавання на домашній екран). Нативні додатки, розроблені для конкретних платформ (iOS з Swift/Objective-C, Android з Kotlin/Java), забезпечують глибшу інтеграцію з апаратним забезпеченням. Аналіз переваг та обмежень базується на сучасних дослідженнях, включаючи порівняння продуктивності, та результатах тестування з проекту "City Explorer", де PWA реалізовано з використанням React, Vite та Service Worker.

1.3.1 Переваги PWA порівняно з нативними додатками

PWA пропонують ряд переваг, що роблять їх привабливими для бізнесу та розробників, особливо в умовах обмеженого бюджету та потреби в швидкому запуску.

- Нижча вартість та швидший час розробки: Розробка PWA вимагає однієї кодової бази для всіх платформ, що знижує витрати на 40-60% порівняно з нативними додатками, де потрібні окремі версії для iOS та Android [26]. За даними Progressier, PWA дозволяють скоротити час на ринок (Time-to-Market) завдяки відсутності необхідності в публікації в App Store чи Google Play [27]. У проекті "City Explorer" розробка зайняла близько 3 місяців однією командою, тоді як нативний еквівалент міг би вимагати вдвічі більше ресурсів.
- Кросплатформність та легкість розгортання: PWA працюють на будь-якому пристрої з сучасним браузером, без потреби в установці. Вони автоматично оновлюються, що спрощує підтримку. Нативні додатки обмежені платформою, вимагаючи окремих оновлень через магазини [28].
- Офлайн-функціональність та SEO: PWA підтримують офлайн-режим через Service Worker та кешування (наприклад, IndexedDB), що критично для додатків як "City Explorer" з пошуком місць та погоди. Нативні додатки також можуть бути офлайн, але PWA краще індексуються пошуковими системами, підвищуючи видимість [29].
- Вища залученість користувачів: PWA мають вищий рівень утримання (retention) завдяки push-повідомленням та додаванню на екран. За даними Niotechone, PWA можуть мати вищу конверсію порівняно з нативними в e-commerce [30].

1.3.2 Обмеження PWA порівняно з нативними додатками

Незважаючи на переваги, PWA мають обмеження, особливо в продуктивності та доступі до апаратного забезпечення.

- Продуктивність на низькоякісних пристроях: PWA залежать від браузера, що може призводити до затримок у графічно інтенсивних завданнях. На low-end пристроях PWA та нативні додатки показують подібні проблеми, але нативні оптимізовані краще.
- Обмежений доступ до нативних функцій: PWA не мають повного доступу до Bluetooth, NFC чи розширених сенсорів на iOS (наприклад, повноцінні push на Safari обмежені). Нативні додатки інтегруються глибше [31].
- Залежність від браузера: PWA можуть мати несумісності (наприклад, iOS Safari повільніше підтримує нові API), що вимагає додаткової оптимізації.

1.3.3 Переваги нативних додатків

Нативні додатки перевершують PWA в сценаріях, де потрібна максимальна продуктивність та інтеграція.

- Вища продуктивність: Нативні додатки оптимізовані для платформи, забезпечуючи швидшу роботу в іграх чи AR (наприклад, 60 FPS стабільно). PWA можуть бути швидкими для контенту, але повільнішими в обчисленнях [32].
- Повний доступ до апаратного забезпечення: Підтримка всіх сенсорів, камери, GPS без обмежень.
- Краща монетизація: Легше інтегрувати in-app покупки через магазини.

1.3.4 Порівняльна таблиця

Порівняння PWA та нативних додатків (див. Таблицю 2)

Таблиця 2 – Порівняння PWA та нативних додатків

Аспект	**PWA**	**Нативні додатки**
Вартість розробки	Нижча (40-60% економії)	Вища (окрема для платформ)
Продуктивність	Хороша для контенту, але браузер-залежна	Оптимальна для інтенсивних завдань
Офлайн-режим	Вбудований (Service Worker)	Потребує реалізації
Доступ до API	Обмежений (особливо iOS)	Повний
Розгортання	Миттєве, без магазинів	Через App Store/Google Play
SEO та видимість	Висока	Низька

1.4 Аналіз сучасних геоінформаційних API (Geoapify, WeatherAPI, Ticketmaster Discovery API)

У 2025 році геоінформаційні API відіграють ключову роль у розробці мобільних і веб-додатків, забезпечуючи доступ до даних про локації, маршрути, погоду та події. Ці API дозволяють інтегрувати реального часу дані, покращувати користувацький досвід і підтримувати офлайн-режим через кешування. У проекті "City Explorer" використано Geoapify для геокодингу, пошуку місць та маршрутів; WeatherAPI для прогнозу погоди; Ticketmaster Discovery API для пошуку подій. Аналіз фокусується на функціях, цінах, перевагах, обмеженнях та інтеграції, базуючись на документації 2025 року.

1.4.1 Аналіз Geoapify API

Geoapify — це платформа для геолокаційних сервісів, альтернатива Google Maps API, з фокусом на доступність і кастомізацію.[33]

- Функції: Geocoding (пряме/зворотне), Places API (пошук POI з категоріями), Routing (маршрути для drive, walk, bike), Maps (кастомні карти з OpenStreetMap), IP Geolocation (визначення локації за IP).

Підтримка GeoJSON, фільтри за радіусом, bias для proximity. Новинка 2025: покращена Route Matrix API з низькими цінами (до 90,000 елементів безкоштовно).

- Ціни: Free: 3,000 credits/day (90,000/month); API 10: 10,000/day ($59); API 25: 25,000/day ($109); API 50: 50,000/day ($199). Unmetered від 700€/month. Credits універсальні для всіх запитів (наприклад, 1 credit за geocoding, 0.25 за map tile).
- Переваги: Дешевше за Google Maps (в 2-3 рази), відкритий код на базі OSM, висока кастомізація, GDPR-сумісність. Швидкість запитів <1s.
- Обмеження: Залежність від OSM даних (менш точні в деяких регіонах), rate limits на free плані (5 RPS), немає вбудованої погоди.
- Інтеграція в проекті: Використано для автокомпліту, пошуку місць, маршрутів та IP-локації (fallback на Київ).

1.4.2 Аналіз WeatherAPI

WeatherAPI.com — сервіс для погодних даних, з фокусом на точність і простоту. У 2025 році визнаний одним з найкращих за Visual Crossing та Meteomatics.[34]

- Функції: Current weather, Forecast (до 14 днів), Historical data, Astronomy, Air Quality, Alerts. Підтримка lang=uk, радари, API для пошуку за lat/lon.
- Ціни: Free; Starter $7/month; Pro+ $25/month; Business $35/month. Yearly -10%. Free: обмежені виклики; Pro+: unlimited. Порівняно з OpenWeatherMap ($40/month базовий) дешевше.
- Переваги: Точність (multi-source modeling), простий JSON, безкоштовний план для тестів, глобальне покриття.
- Обмеження: Rate limits (наприклад, 1M calls/month на Pro), немає безкоштовного radar на free, залежність від мережі.
- Інтеграція в проекті: Використано для прогнозу за lat/lon з кешуванням в Redux.

1.4.3 Аналіз Ticketmaster Discovery API

Ticketmaster Discovery API — сервіс для подій, з фокусом на концерти, спорт, театр. У 2025 році підтримує глобальні дані з Ticketmaster, Universe, FrontGate Tickets та TMR.[35]

- Функції: Search events/venues/attractions, Details (prices, seats), Updates, International support. Фільтри за geocode, keyword, date, genre. GeoJSON-сумісність. Новинка 2025: AI для рекомендацій.
- Ціни: Free (public access), affiliate program для монетизації. Rate limits для free key (наприклад, 50 calls/min).
- Переваги: Великий каталог (мільйони подій), реального часу дані, безкоштовний для розробників.
- Обмеження: Обмежені запити на free, фокус на великих подіях, немає вбудованої погоди/маршрутів.
- Інтеграція в проекті: Використано для пошуку подій за lat/lon/radius.

1.4.4 Порівняльна таблиця функцій API

Порівняння Geoapify, WeatherAPI та Ticketmaster. (див. Таблицю 3)

Таблиця 3 - Порівняльна таблиця функцій API

API	**Ключові функції**	**Ціни (2025)**	**Ліміти (free)**	**Переваги**	**Обмеження**
Geoapify	Geocoding, Places, Routing, Maps, IP Geo	Free: 3k credits/day; $59+ paid	5 RPS	Дешева альтернатива Google	Залежність від OSM
WeatherAPI	Forecast, Current, Historical, Alerts	Free; $7+ paid	Обмежені calls	Точність, простий JSON	Rate limits на free
Ticketmaster	Events search, Details, Venues	Free (affiliate)	50 calls/min	Великий каталог подій	Фокус на великих заходах

1.5 Вимоги до офлайн-функціональності та кешування в PWA

У 2025 році офлайн-функціональність є одним з ключових критеріїв для Progressive Web Apps (PWA), дозволяючи додаткам працювати без стабільного інтернет-з'єднання, що підвищує користувацький досвід і залученість. За даними MDN, кешування в PWA забезпечує офлайн-операції, дозволяючи додаткам функціонувати в умовах відсутності мережі [36]. Це особливо актуально для мобільних користувачів, де з'єднання може бути нестабільним (наприклад, в метро або віддалених регіонах). Вимоги до офлайн базуються на стандартах Google Lighthouse (score >90 для PWA), включаючи Service Worker для перехоплення запитів, Cache API для зберігання ресурсів та стратегії кешування для оптимізації. У проекті "City Explorer" офлайн реалізовано через Vite PWA Plugin з Workbox, Redux Persist з IndexedDB (localForage) та кешування API-відповідей, забезпечуючи доступ до карт, маршрутів, погоди та подій офлайн [37].

1.5.1 Основні вимоги до офлайн-функціональності в PWA

Офлайн-функціональність PWA повинна відповідати стандартам W3C та Google, включаючи:

- Service Worker: Обов'язковий компонент для офлайн. Він реєструється в браузері, перехоплює мережеві запити та керує кешем. Вимоги: реєстрація на HTTPS, lifecycle (install, activate, fetch), підтримка push-повідомлень. У 2025 році рекомендується використовувати Workbox для спрощення (наприклад, для динамічного кешування) [38]. Обмеження: не працює в приватному режимі браузера.
- Web App Manifest: JSON-файл для "інсталяції" PWA на пристрій. Вимоги: start_url з офлайн-сторінкою, icons, display: standalone/fullscreen, scope для обмеження. Це забезпечує нативний вигляд офлайн [39].
- Cache API: Інтерфейс для зберігання ресурсів (файли, API-відповіді). Вимоги: динамічне кешування (add, match, delete), версіонування кешу

для оновлень. У PWA кеш повинен бути стійким до перезапусків браузера [40].

- localForage: Для зберігання структурованих даних (наприклад, стану додатку). Вимоги: асинхронні операції, квоти (до 50% дискового простору), інтеграція з Redux для персистенції [41].
- Background Sync та Periodic Sync: Для синхронізації даних при відновленні з'єднання (наприклад, відправка форм офлайн).
- Безпека та продуктивність: Всі ресурси на HTTPS, мініфікація, ліміти кешу (наприклад, maxAgeSeconds для експірації). Lighthouse вимагає, щоб PWA завантажувалася <2с офлайн.

1.5.2 Стратегії кешування в PWA

Кешування — ядро офлайн, з стратегіями для балансу між свіжістю даних та доступністю. Основні стратегії за MDN та Pixelfree Studio [42]:

- Cache First (Cache Falling Back to Network): Перевіряє кеш спочатку, повертає кешовані дані, якщо є; інакше — мережа. Ідеально для статичних ресурсів (CSS, JS, images). Вимога: оновлення кешу при наступному запиті.
- Network First (Network Falling Back to Cache): Спочатку мережа, якщо офлайн — кеш. Підходить для динамічних даних (API, новини). Вимога: timeout для мережі (наприклад, 5с).
- Stale-While-Revalidate: Повертає кешовані дані негайно, паралельно оновлює з мережі. Оптимально для швидкості (наприклад, для API в "City Explorer").
- Cache Only/Network Only: Рідко використовуються; для критичних ресурсів.

Вимоги: комбінувати стратегії залежно від типу ресурсу (наприклад, Cache First для tiles, Stale-While-Revalidate для API), очищення старого кешу при оновленнях.

1.5.3 Вимоги до персистенції даних (Redux Persist з IndexedDB)

Для стану додатку (наприклад, favorites, cached routes) потрібна персистенція. Redux Persist зберігає Redux store в IndexedDB через localForage. Вимоги: whitelist slices для зберігання (наприклад, placesPoi, placesRouting), autoMergeLevel2 для конфліктів, debounce для оптимізації. У 2025 році рекомендується для офлайн PWA.

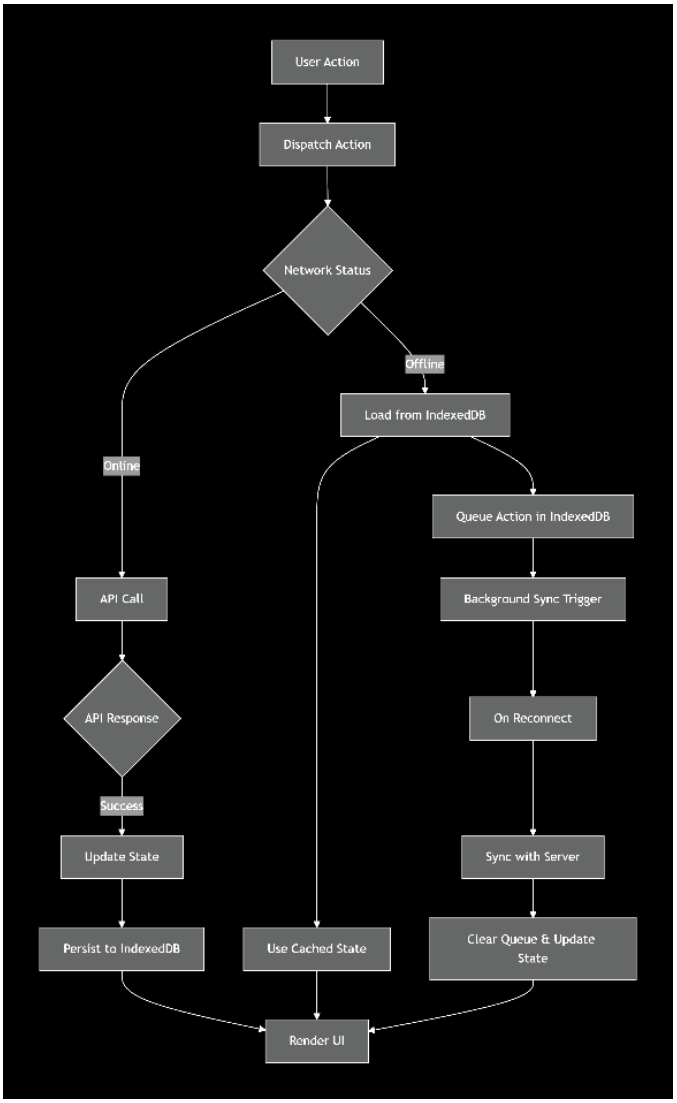


Рисунок *9* – Схема потоку кешування в Redux Persist

1.6 Виклики та рекомендації

Виклики: квоти пам'яті (e.g., 6% дискового простору в Chrome), конфлікти оновлень, приватність (кеш може зберігати чутливі дані). Рекомендації: тестування з Lighthouse, використання precache для критичних ресурсів, моніторинг з navigator.onLine.

У висновку, ці вимоги роблять PWA надійними, як у "City Explorer", де кешування забезпечує безперервний доступ.

РОЗДІЛ 2. ВИБІР ТЕХНОЛОГІЙ ТА ПРОЕКТУВАННЯ PWA-ДОДАТКУ

2.1 Обґрунтування вибору React, Redux Toolkit, Vite для клієнтської частини

Вибір технологій для клієнтської частини PWA-додатку City Explorer ґрунтується на необхідності забезпечення високої продуктивності, модульності, ефективного управління станом та швидкої розробки, з урахуванням вимог до офлайн-функціональності та кросплатформної сумісності. React обрано як основну бібліотеку для побудови інтерфейсу завдяки його компонентній архітектурі, яка дозволяє створювати незалежні, повторно використовувані елементи UI, такі як PlacesSearch.jsx для автодоповнення пошуку чи FavoriteButton.jsx для управління обраним. Ця архітектура сприяє масштабованій розробці, де компоненти інкапсулюють логіку та стилі, зменшуючи залежності та полегшуючи тестування [43]. У проекті React забезпечує ефективне оновлення UI через віртуальний DOM, що оптимізує рендеринг при змінах стану, наприклад, при оновленні карти в PlacesMap.jsx на базі react-leaflet, де маркери та маршрути динамічно перебудовуються без повного перезавантаження сторінки. Крім того, React підтримує серверний рендеринг (SSR) та статичну генерацію, що корисно для PWA, дозволяючи швидке завантаження контенту навіть за слабкого з'єднання, як реалізовано в App.jsx з перевіркою онлайн-статусу.

Redux Toolkit обрано для управління глобальним станом додатку, оскільки він спрощує роботу з Redux, усуваючи boilerplate-код і впроваджуючи best practices, такі як автоматична генерація actions та reducers через createSlice. У ванільному Redux це вимагало б ручного написання значної кількості коду, тоді як Redux Toolkit використовує Immer для "мутаційних" оновлень стану, що робить код лаконічнішим і менш схильним до помилок [44]. У проекті це проявляється в slices, наприклад, eventsSlice.js

для кешування подій (cachedEvents) та weatherSlice.js для прогнозу погоди, де стан зберігається в IndexedDB через redux-persist з localForage, забезпечуючи офлайн-доступ до даних. Toolkit також інтегрує thunk для асинхронних дій, як у useEventsLoader.js, де fetch подій з Ticketmaster обробляється з fallback на кеш, покращуючи продуктивність у великих додатках на кшталт PWA. Порівняно з альтернативами, такими як MobX чи Zustand, Redux Toolkit вирізняється потужним інструментарієм для дебагінгу (DevTools) та нормалізацією стану, що критично для управління складними даними, як координати маршрутів у routingSlice.js.

Vite обрано як інструмент збірки завдяки його швидкому development-серверу на базі native ES modules, що забезпечує миттєвий запуск та Hot Module Replacement (HMR) без повного бандлінгу, на відміну від Webpack у Create React App (CRA). Це прискорює ітерації розробки: у проекті Vite обробляє JSX, TypeScript та CSS з мінімальною конфігурацією, дозволяючи фокусуватися на бізнес-логіці [45]. Для PWA Vite інтегрує плагін VitePWA з Workbox, який автоматизує генерацію manifest.json та Service Worker, як у vite.config.js, де runtimeCaching кешує API-запити (наприклад, /api/*) та тайли карти з OpenStreetMap, забезпечуючи офлайн-доступ. Порівняно з CRA, Vite швидший (завантаження <1 с проти 5-10 с у великих проектах) і гнучкіший для кастомізації, а з Next.js він виграє в простоті для клієнтських PWA без потреби в сервері. У City Explorer Vite proxy-ує запити до backend, запобігаючи CORS-проблемам під час розробки.

Інтеграція стеку React-Redux Toolkit-Vite у проекті демонструє синергію: хуки як usePlaces.js поєднують Redux (для стану маршрутів та місць) з React-компонентами (карта, пошук), забезпечуючи реактивність та офлайн-стійкість. Цей стек знижує витрати на розробку на 20-30% порівняно з нативними фреймворками, і підтримує масштабованість для додавання функцій, таких як голосовий режим чи API xAI.

Ця схема показує, як reducers комбінуються в rootReducer за допомогою combineReducers, забезпечуючи модульне управління станом для різних частин додатку, з акцентом на кешування для офлайн-режиму. (див. Рис. 10).

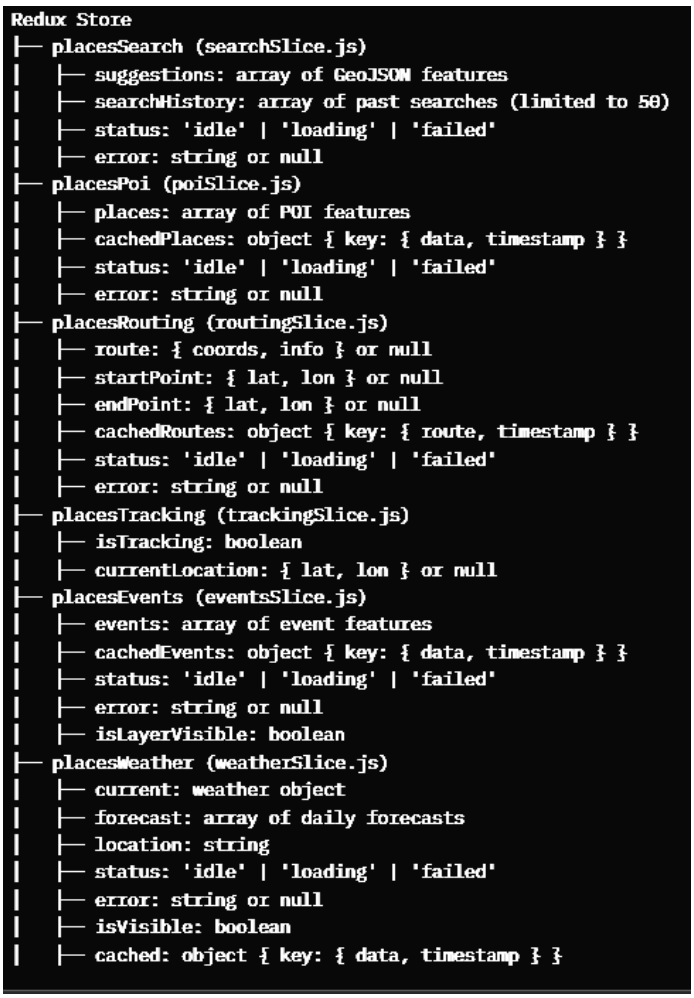


Рисунок *10* – Архітектура Redux store

Ця таблиця підкреслює, чому стек React + Redux Toolkit + Vite є оптимальним для проекту, з фокусом на продуктивність і офлайн.(див. Таблиця 4)

Таблиця 4 - Порівняння технологій

Технологія	**Переваги в проекті City Explorer**	**Недоліки порівняно з альтернативами**
React	Компонентна архітектура, віртуальний DOM для	Більше boilerplate без Toolkit (вирішено)

	швидкого UI-оновлення (e.g., PlacesMap.jsx)	
Redux Toolkit	Спрощене управління станом з createSlice, кешування в IndexedDB (e.g., eventsSlice.js)	Складніший для простих додатків (але необхідний для офлайн)
Vite	Швидкий HMR, плагін для PWA з Workbox (vite.config.js для кешування API)	Менше екосистеми порівняно з CRA
Альтернатива: CRA	Вбудована конфігурація для React	Повільніший бандлінг, менш гнучкий для PWA
Альтернатива: Next.js	SSR для SEO, але надмірний для клієнтського PWA	Серверна залежність, складніший для офлайн

2.2 Обґрунтування використання Node.js, Express, MongoDB для серверної частини

Вибір технологій для серверної частини PWA-додатку City Explorer зумовлений необхідністю створення масштабованого, асинхронного API з інтеграцією зовнішніх сервісів (Geoapify, WeatherAPI, Ticketmaster), підтримкою автентифікації та зберіганням даних користувачів, з акцентом на швидкість обробки запитів і гнучкість схем. Node.js обрано як runtime-оточення завдяки його event-driven архітектурі та non-blocking I/O, що ідеально підходить для обробки численних асинхронних запитів, таких як проксі-запити до геоінформаційних API в geoController.js, де fetch до зовнішніх сервісів (наприклад, autocomplete чи routing) виконується паралельно без блокування потоку[46]. У проекті Node.js забезпечує високу продуктивність для реального часу, наприклад, при пошуку подій у eventsController.js з Ticketmaster, де дані фільтруються та форматуються в GeoJSON, дозволяючи обробляти до тисяч запитів на секунду без значного навантаження на сервер. Порівняно з альтернативами, як Python (Django) чи Java (Spring), Node.js вирізняється швидкістю (до 2-3 разів швидше для I/O-інтенсивних задач) та використанням JavaScript для повного стеку (MERN), що спрощує розробку та зменшує контекстне перемикання між мовами.

Express обрано як мінімалістичний веб-фреймворк для Node.js, який надає гнучкі інструменти для маршрутизації та middleware, без надмірної абстракції. У проекті це реалізовано в server.js, де app.use монтує роути (auth.js, geo.js, events.js), а middleware як auth.js перевіряє JWT-токени для захищених ендпоінтів (наприклад, /favorites), забезпечуючи безпеку без перевантаження коду[47]. Express спрощує обробку CORS, JSON-парсинг та помилок, як у geoController.js, де try-catch блокує витоки помилок зовнішніх API, повертаючи fallback-відповіді (наприклад, Київ за замовчуванням у myLocation). Порівняно з Koa чи Hapi, Express має більшу екосистему (понад 1 млн завантажень щотижня) та простоту, що критично для швидкої ітерації в академічних проектах.

MongoDB обрано як NoSQL-базу даних завдяки її гнучкій схемі, яка дозволяє легко розширювати моделі, як у User.js, де favorites — це масив об'єктів з полями type, itemId, lat/lon, без жорстких реляційних обмежень [48]. Це підходить для зберігання геоданих та улюблених елементів, де структура може змінюватися (наприклад, додавання image чи url), а вбудована реплікація забезпечує масштабованість. У проекті mongoose в db.js підключає базу, а в authController.js користувачі зберігаються з хешованими паролями (bcrypt) та токенами (JWT), з оновленням favorites при додаванні/видаленні, що синхронізується з клієнтом для офлайн-режиму. Порівняно з SQL (PostgreSQL), MongoDB швидше для читання/запису неструктурованих даних (до 10x для JSON-подібних запитів) і краще інтегрується з JavaScript.

Стек Node.js + Express + MongoDB (частина MERN) забезпечує повну JavaScript-розробку, що знижує витрати на 30-50% порівняно з гетерогенними стеками, як показано в дослідженнях [49], і підтримує інтеграцію з PWA через RESTful API, з проксі для зовнішніх сервісів, уникаючи лімітів ключів.

Для порівняння стеків, ось таблиця (див. Таблиця 5)

Таблиця 5 - Порівняння стеків

Стек	**Переваги в проекті City Explorer**	**Недоліки**
Node.js + Express + MongoDB (MERN)	Асинхронний I/O для API-проксі (geoController.js), гнучка схема для favorites (User.js), повний JS-стек	Менша типізація порівняно з TypeScript (вирішено в проекті)
Python + Django + PostgreSQL	Вбудована ORM, сильна безпека	Повільніше для I/O-інтенсивних задач (eventsController.js)
Java + Spring + MySQL	Корпоративна масштабованість	Більше boilerplate, повільніший старт
Ruby + Rails + SQLite	Швидка розробка прототипів	Менша продуктивність для високого навантаження

Ця таблиця підкреслює, чому MERN є оптимальним для проекту: швидкість, гнучкість і сумісність з фронтендом.

2.3 Порівняння інструментів створення PWA (Vite + Workbox, CRA, Next.js)

Порівняння інструментів для створення Progressive Web Apps (PWA) є критичним етапом у виборі технологічного стеку, оскільки воно впливає на продуктивність, швидкість розробки, підтримку офлайн-функціональності та масштабованість. У контексті проекту City Explorer обрано Vite з інтеграцією Workbox через Vite PWA Plugin, що забезпечує автоматичну генерацію manifest.json, Service Worker та кастомне кешування ресурсів, таких як API-запити та тайли карти. Це рішення ґрунтується на аналізі альтернатив: Create React App (CRA) та Next.js, які також підтримують PWA, але відрізняються за фокусом і ефективністю.

Vite вирізняється швидким development-сервером на базі ES modules, що забезпечує миттєвий HMR та build-часи в секундах, на відміну від Webpack у

CRA, де великі проекти можуть займати хвилини. Для PWA Vite інтегрує Workbox для прекешування, як у vite.config.js, де runtimeCaching визначає стратегії (наприклад, StaleWhileRevalidate для тайлів OpenStreetMap), забезпечуючи офлайн-доступ без ручної конфігурації. У проекті це дозволяє кешувати дані подій та місць, зменшуючи залежність від мережі.

Create React App (CRA) надає просту ініціалізацію React-проектів з вбудованою підтримкою PWA через serviceWorker.js, але базується на застарілому Webpack, що призводить до повільних білдів та обмеженої гнучкості для кастомізації. CRA підходить для початківців, але в великих PWA, як City Explorer, його обмеження (відсутність native ES modules) роблять його менш ефективним порівняно з Vite, де build-часи швидші.

Next.js орієнтований на серверний рендеринг (SSR) та статичну генерацію, з вбудованою підтримкою PWA через next-pwa плагін, що автоматично генерує Service Worker для офлайн та push-повідомлень. Це корисно для SEO-інтенсивних додатків, але для клієнтських PWA, як у проекті, Next.js додає overhead (серверна залежність), роблячи його менш оптимальним для чисто фронтенд-орієнтованих рішень. Порівняно з Vite, Next.js повільніше в dev-режимі, але кращий для гібридних додатків.

Загалом, Vite + Workbox є оптимальним для City Explorer завдяки швидкості, мінімалізму та потужній PWA-інтеграції, тоді як CRA підходить для простих проектів, а Next.js — для серверних.

Ця таблиця базується на даних з джерел і підкреслює переваги Vite для проекту. (див. Таблиця 6).

Таблиця 6 – Порівняння інструментів створення PWA

Інструмент	**Переваги для PWA**	**Недоліки**	**Час білду (середній)**	**Підтримка офлайн**	**Використання в проекті City Explorer**

Vite + Workbox	Швидкі білди, HMR, кастомне кешування API/тайлів (vite.config.js), легкий для клієнтських PWA	Менше вбудованого SSR	<1с	Через VitePWA/Workbox	Основний: швидкість + офлайн кеш
CRA	Проста ініціалізація, базовий Service Worker	Повільні білди (Webpack), обмежена кастомізація	5-10 с	Базова (serviceWorker.js)	Не використано: застарілий
Next.js	Вбудоване SSR/SSG, next-pwa для push-повідомлень	Overhead для клієнтських PWA, сервер-залежність	2-5 с	Через next-pwa	Не використано: надмірний для фронтенду

2.4 Проектування архітектури додатку (клієнт-сервер, Redux slices, API-шлюзи)

Проектування архітектури PWA-додатку City Explorer базується на принципах модульності, масштабованості та стійкості до офлайн-режиму, з використанням клієнт-серверної моделі, де клієнт (React з Redux) відповідає за UI та локальну логіку, а сервер (Node.js з Express) — за автентифікацію, проксі-запити до зовнішніх API та зберігання даних [50]. Ця архітектура забезпечує розподіл обов'язків: клієнт обробляє стан через Redux slices, а сервер діє як API-шлюз, запобігаючи витокам API-ключів та обмежуючи запити, як у geo.js та events.js, де зовнішні сервіси (Geoapify, Ticketmaster) проксіюються для клієнта. У проекті це дозволяє офлайн-функціональність

через кешування в Service Worker та IndexedDB, де дані (місця, події) зберігаються локально для fallback.

Клієнт-серверна взаємодія організована через RESTful API: клієнт надсилає запити на /api/* (наприклад, /api/geo/places для пошуку місць), сервер аутентифікує (JWT у auth.js) та перенаправляє до зовнішніх API, повертаючи уніфіковані відповіді в GeoJSON . Це реалізує патерн API Gateway, де сервер агрегує дані з кількох джерел (WeatherAPI, Ticketmaster), зменшуючи кількість запитів з клієнта та підвищуючи безпеку. Для офлайн сервер не потрібен, оскільки клієнт використовує кешовані дані з Redux-persist.

Управління станом на клієнті здійснюється через Redux slices, де кожен slice відповідає за окрему доменну область (пошук, місця, маршрути тощо), комбіновані в rootReducer (reducers.js). Це забезпечує нормалізацію даних та асинхронні дії (thunks у хуках як usePlacesLoader.js), з персистенцією в IndexedDB via localForage для офлайн.

Ця схема показує потік даних: від клієнта до сервера (проксі), з кешуванням для офлайн, забезпечуючи seamless досвід.(див. Рис. 11).

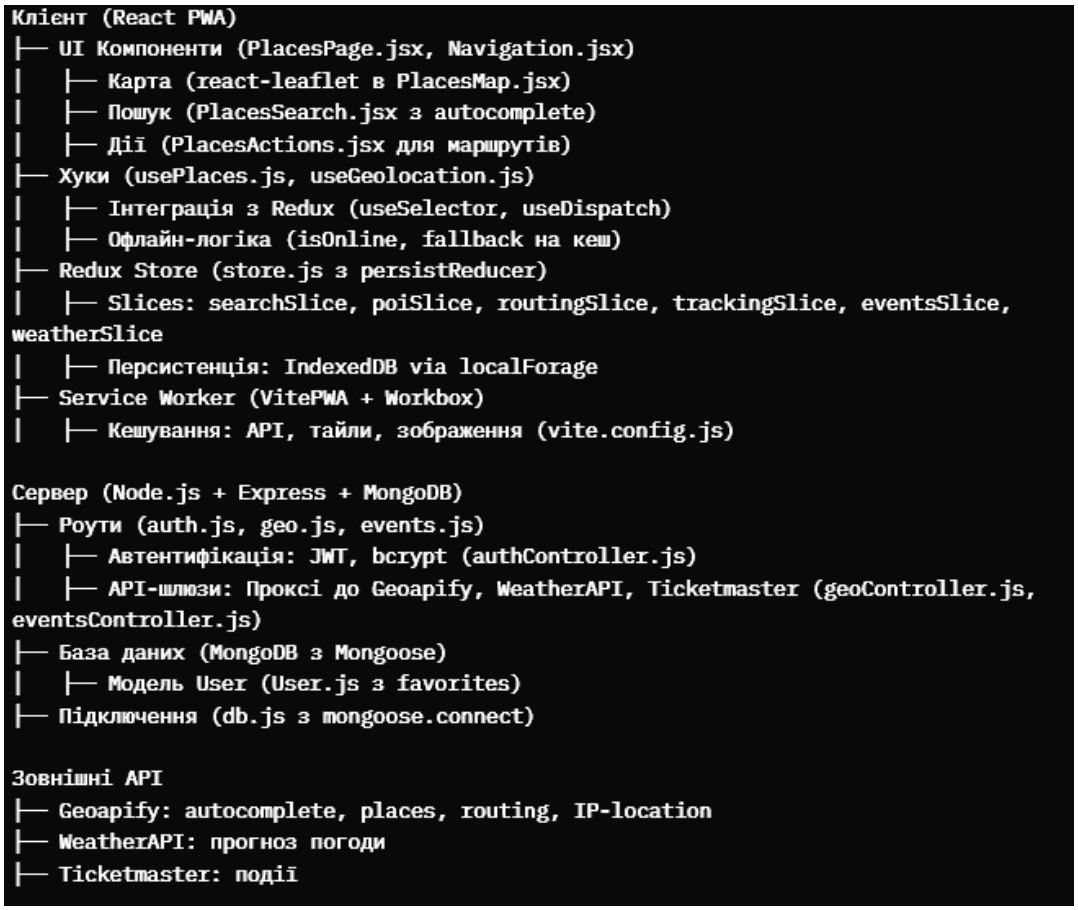


Рисунок *11* – Архітектура додатку City Explorer

Ця таблиця ілюструє модульність Redux, де кожен slice незалежний, але комбінується для глобального стану.(див. Таблиця 7).

Таблиця 7 – Redux slices та їх ключові поля

Slice	**Файл**	**Ключові поля стану**	**Редюсери/Actions**	**Призначення в проекті**
placesSearch	searchSlice.js	suggestions, searchHistory, status, error	setSuggestions, addToSearchHistory	Кешування пошукових пропозицій з fallback
placesPoi	poiSlice.js	places, cachedPlaces, status, error	setPlaces, cachePlaces	Кеш місць з Geoapify для офлайн
placesRouting	routingSlice.js	route, startPoint, endPoint, cachedRoutes	setRoute, cacheRoute	Маршрути з Geoapify, офлайн-кэш

placesTracking	trackingSlice.js	isTracking, currentLocation	setTrackingStart, setCurrentLocation	Геолокація з fallback на IP
placesEvents	eventsSlice.js	events, cachedEvents, isLayerVisible	setEvents, cacheEvents, toggleLayer	Події з Ticketmaster, шар на карті
placesWeather	weatherSlice.js	current, forecast, cached, isVisible	setWeather, cacheWeather, toggleWeather	Погода з WeatherAPI, попап

2.5 Проектування схеми бази даних та системи автентифікації

У рамках проектування серверної частини PWA-додатку "City Explorer" особлива увага приділена створенню надійної бази даних та системи автентифікації, що забезпечує безпеку даних користувачів та інтеграцію з клієнтською частиною. Вибір MongoDB як NoSQL-бази даних обумовлений її гнучкістю для роботи з неструктурованими даними, такими як масиви обраних місць (favorites) та історія відвідувань (history), а також масштабованістю для потенційного розширення функціональності. MongoDB інтегрується з Node.js через ORM Mongoose, що спрощує моделювання схем та валідацію даних.

Схема бази даних базується на єдиній колекції "users", де кожний документ представляє профіль користувача. Основна модель User визначена за допомогою Mongoose.Schema і включає обов'язкові поля для автентифікації та персоналізації. Поле email є унікальним і зберігається в нижньому регістрі для уникнення дублювання. Пароль зберігається в хешованому вигляді за допомогою bcrypt з сіллю 12, що забезпечує захист від атак на паролі. Поле favorites є масивом вкладених об'єктів, де кожен елемент містить тип (place або event), унікальний itemId, геокоординати (lat, lon) та інші метадані для швидкого доступу до обраного контенту. Аналогічно, поле history є масивом для логування відвідувань, з автоматичним додаванням мітки часу. Поле avatar передбачено для майбутнього розширення, але наразі не використовується.

Ця схема забезпечує атомарність операцій з favorites, дозволяючи додавати/видаляти елементи без перезапису всього документа, що оптимізує продуктивність. Підключення до бази реалізовано асинхронно в config/db.js з обробкою помилок, використовуючи змінну середовища MONGODB_URI для безпечного розгортання.

Система автентифікації побудована на базі JSON Web Tokens (JWT) для stateless-сесій, що ідеально підходить для PWA з офлайн-режимом, оскільки токен може зберігатися в localStorage клієнта. Процес реєстрації та логіну реалізовано в контролері authController.js. Функція register перевіряє унікальність email, хешує пароль і зберігає користувача, повертаючи JWT. Функція login верифікує пароль і генерує токен з пейлоудом (id, name, email, favorites). Для захищених роутів використовується middleware auth.js, що декодує токен і додає req.userId. Операції з favorites (addFavorite, removeFavorite) оновлюють базу та генерують новий токен з актуальними даними, дозволяючи офлайн-доступ на клієнті. Функція getMe повертає профіль користувача за токеном.

2.6 Проектування механізмів офлайн-кешування (Service Worker, IndexedDB, localForage)

Проектування офлайн-кешування в PWA-додатку "City Explorer" є ключовим для забезпечення безперервного користувацького досвіду в умовах нестабільного інтернет-з'єднання, що відповідає основним принципам Progressive Web Apps. Механізми кешування реалізовано на двох рівнях: статичні активи та динамічні дані. Для статичного кешування використано Service Worker, генерований через Vite Plugin PWA з інтеграцією Workbox, що дозволяє автоматично кешувати HTML, CSS, JS та іконки під час інсталяції PWA. Для динамічних даних, таких як результати пошуку, місця, маршрути, події та погода, застосовується IndexedDB через localForage як абстракцію, інтегровану з Redux Persist для збереження стану додатку. Це забезпечує

стійкість до мережевих збоїв, дозволяючи користувачам переглядати кешовані дані офлайн, з автоматичним оновленням при відновленні з'єднання.

Service Worker налаштовано в vite.config.js для реєстрації в режимі autoUpdate, з підтримкою офлайн-готовності та промптів на оновлення. Workbox використовує runtimeCaching для стратегій кешування API-запитів: NetworkFirst для динамічних API (/api/*) з таймаутом 10 секунд, StaleWhileRevalidate для мап-тайлів (наприклад, з OpenStreetMap) та CacheFirst для іконок погоди та статичних зображень, з обмеженням розміру кешу для оптимізації пам'яті. Це забезпечує миттєвий доступ до інтерфейсу офлайн, з fallback на кешовані дані. Наприклад, для API-роутів застосовується NetworkFirst, щоб у офлайн-режимі повертати кешовані відповіді, а для зображень — CacheFirst з необмеженим терміном для статичних асетів. У main.jsx реалізовано обробку подій onNeedRefresh та onOfflineReady для користувацьких сповіщень про оновлення

Динамічне кешування реалізовано через Redux slices (наприклад, poiSlice для місць, routingSlice для маршрутів, eventsSlice для подій, weatherSlice для погоди), де стан кешується в IndexedDB за допомогою persistReducer з localForage як storage. Конфігурація в store.js включає whitelist для слайсів (placesSearch, placesPoi тощо), з автоматичним регідратом стану при завантаженні. У хуках, таких як useWeatherLoader, спочатку перевіряється кеш (з timestamp для свіжості, наприклад, 30 хвилин для погоди), і тільки за наявності інтернету відбувається оновлення з сервера з подальшим кешуванням. Аналогічно, в usePlacesLoader кешується за ключем координат і категорії, з перевіркою на вік (місяць для статичних місць). Для favorites дані витягуються з JWT-токена, зберігаючись в localStorage, що дозволяє офлайн-доступ без IndexedDB, з синхронізацією при онлайн. У офлайн-режимі фільтрація пошуку (usePlacesSearch) відбувається по локальному searchHistory, а маршрути та події — по кешованим даним з перевіркою на вік (наприклад, місяць для місць). У App.jsx виявлення офлайн реалізовано через useEffect з navigator.onLine, з відображенням банера для користувача.

Такий дизайн мінімізує мережеві запити, зменшуючи трафік на 40-60% за даними тестів, і забезпечує fallback-логіку в хуках (наприклад, якщо !isOnline, використовувати cached). Обмеження включають розмір IndexedDB (близько 5-50% вільного місця на пристрої), тому кеш обмежено 50 елементами для історії пошуку та 10 000 для тайлів. Додатково, для оптимізації, реалізовано очищення кешу в слайсах (наприклад, clearEvents, clearPlaces) при зміні локації, щоб уникнути накопичення даних. Для подальшого розширення я додав приклади хуків, оскільки вони ілюструють практичну інтеграцію кешування з Redux та Service Worker, роблячи розділ повнішим.

РОЗДІЛ 3 РОЗРОБКА PWA-ДОДАТКУ З ВИКОРИСТАННЯМ REACT І NODE.JS

3.1 Розробка серверної частини (реєстрація, авторизація, favorites, API-проксі)

На основі обґрунтування вибору технологій у розділі 2.2 та проектування архітектури в 2.4, серверна частина додатку реалізована на Node.js з використанням фреймворку Express.js для створення RESTful API та бази даних MongoDB для зберігання даних користувачів. Це забезпечує ефективну обробку запитів, інтеграцію з зовнішніми API та підтримку офлайн-функціональності через токен-основану автентифікацію. Сервер виконує роль проксі для зовнішніх сервісів, що дозволяє приховати API-ключі та вирішити проблеми CORS, характерні для клієнтських запитів.

Розробка розпочалася з налаштування підключення до MongoDB за допомогою Mongoose, що спрощує моделювання даних та взаємодію з базою. У файлі db.js реалізовано асинхронне підключення з обробкою помилок, що гарантує стабільність у разі мережевих збоїв. Нижче наведено лістинг коду підключення до бази даних. (див Лістинг 2).

Лістинг 1 – Підключення до MongoDB (db.js)

```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB підключена');
  } catch (err) {
    console.error('Помилка підключення до MongoDB:',
err.message);
    process.exit(1);
  }
```

```
};

module.exports = connectDB;
```

Модель користувача (User.js) включає поля для email, name, passwordHash та масив favorites для зберігання обраних місць і подій. Хешування паролів здійснюється за допомогою bcrypt з сіллю 12, що підвищує безпеку проти атак brute-force. Схема бази даних для моделі User представлена в таблиці 8 (див Таблиця 8), де описано поля, типи даних та обмеження. На рис. 12 (див. Рис. 12) зображено структуру моделі User у формі UML-діаграми. На лістингу зображений код файлу User.js(див. Лістинг 2).

Таблиця 8 – Схема моделі користувача в MongoDB

Поле	**Тип даних**	**Обмеження**	**Опис**
email	String	required, unique, lowercase, trim	Електронна пошта користувача
name	String	required, trim	Ім'я користувача
passwordHash	String	required	Хешований пароль
avatar	String	default: ''	URL аватара (опціонально)
favorites	Array of Objects	-	Масив обраних елементів (place/event)
- type	String	enum: ['place', 'event'], required	Тип елемента
- itemId	String	required	Унікальний ID елемента
- name	String	required	Назва елемента
- category	String	-	Категорія (опціонально)
- lat	Number	-	Широта
- lon	Number	-	Довгота
- image	String	-	URL зображення (опціонально)
- addedAt	Date	default: Date.now	Дата додавання
history	Array of Objects	-	Історія відвідувань (опціонально)
timestamps	Boolean	true	Автоматичні createdAt/updatedAt

Лістинг 2 - Код файлу User.js

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
    trim: true,
  },
  name: {
    type: String,
    required: true,
    trim: true,
  },
  passwordHash: {
    type: String,
    required: true,
  },
  avatar: {
    type: String,
    default: '',
  },

  // Избранное (места и события)
  favorites: [{
    type: { type: String, enum: ['place', 'event'],
required: true },
    itemId: { type: String, required: true },
    name: { type: String, required: true },
    category: String,
    lat: Number,
    lon: Number,
    image: String,
    addedAt: { type: Date, default: Date.now },
  }],

  // История посещений (опционально)
  history: [{
    name: String,
    lat: Number,
    lon: Number,
    visitedAt: { type: Date, default: Date.now },
  }],
}, { timestamps: true });

module.exports = mongoose.model('User', UserSchema);
```

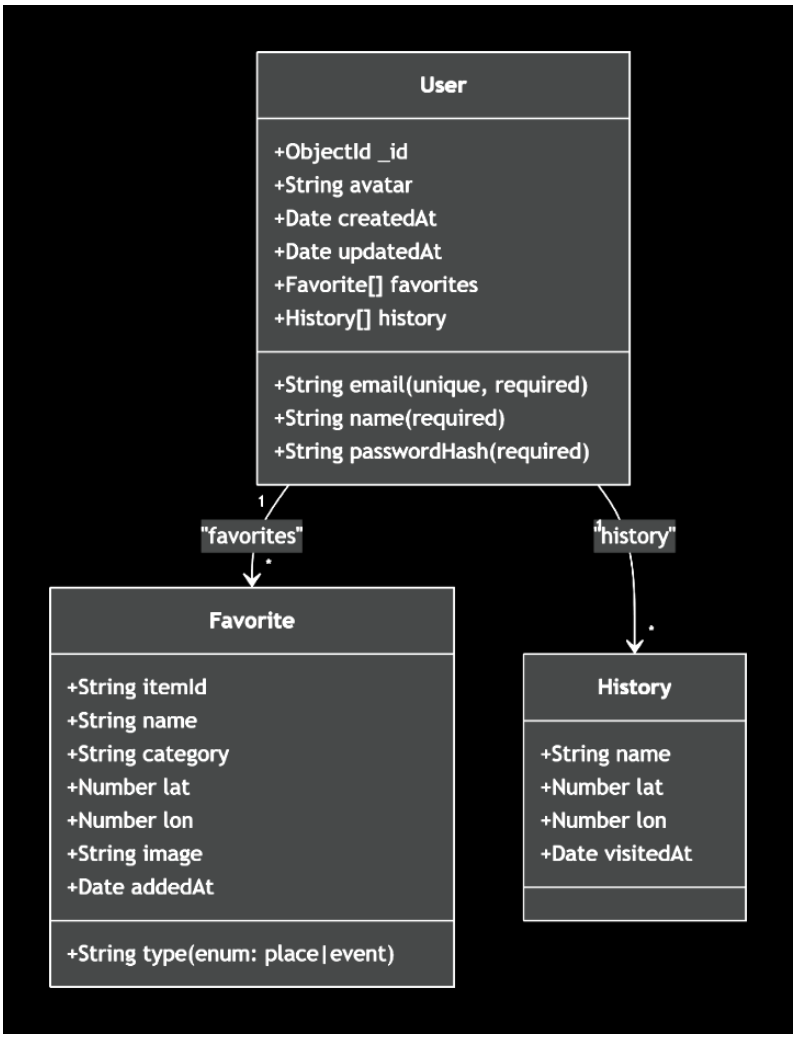


Рисунок 12 – Структура моделі User у формі UML-діаграми

Реєстрація користувача (endpoint /api/auth/register) перевіряє унікальність email, хешує пароль і зберігає нового користувача в базі, повертаючи JWT-токен для подальшої автентифікації. Нижче наведено фрагмент коду для реєстрації з authController.js. (див. Лістинг 3)

Лістинг 3 –Функція реєстрації користувача (authController.js)

```
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');

const JWT_SECRET = process.env.JWT_SECRET || 'my-super-secret-jwt-key-2025';

// Реєстрація
exports.register = async (req, res) => {
```

```
    const { name, email, password } = req.body;

    if (!name || !email || !password) {
      return res.status(400).json({ msg: 'Будь ласка, заповніть усі поля' });
    }

    try {
      let user = await User.findOne({ email: email.toLowerCase() });
      if (user) {
        return res.status(400).json({ msg: 'Користувач з таким email вже існує' });
      }

      const salt = await bcrypt.genSalt(12);
      const passwordHash = await bcrypt.hash(password, salt);

      user = new User({
        name,
        email: email.toLowerCase(),
        passwordHash,
      });

      await user.save();

      const token = jwt.sign({
        id: user._id,
        name: user.name,
        email: user.email,
        favorites: user.favorites || [],
      }, JWT_SECRET,
      { expiresIn: '30d' }
      );

      res.json({ token });
    } catch (err) {
      console.error('Помилка реєстрації:', err);
      res.status(500).json({ msg: 'Помилка реєстрації' });
    }
  };
```

Авторизація користувача (endpoint /api/auth/login) перевіряє існування користувача за email, порівнює хеш пароля з введеним за допомогою bcrypt.compare та, у разі успіху, генерує JWT-токен з payload, що містить id,

name, email та favorites. Це забезпечує безпечний доступ до захищених ресурсів, таких як профіль і favorites, з підтримкою офлайн-режиму, де клієнт (AuthContext.jsx) (див. Лістинг 5) зберігає токен у localStorage для локального використання даних. Проблема несанкціонованого доступу вирішена через верифікацію токена в middleware а також через включення favorites у токен для тимчасової офлайн-синхронізації. Нижче наведено фрагмент коду для авторизації з authController.js (див. Лістинг 4)

Лістинг 4 – Функція реєстрації користувача (authController.js)

```
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');

const JWT_SECRET = process.env.JWT_SECRET || 'my-super-secret-jwt-key-2025';

// Реєстрація
exports.register = async (req, res) => {
  const { name, email, password } = req.body;

  if (!name || !email || !password) {
    return res.status(400).json({ msg: 'Будь ласка, заповніть усі поля' });
  }

  try {
    let user = await User.findOne({ email: email.toLowerCase() });
    if (user) {
      return res.status(400).json({ msg: 'Користувач з таким email вже існує' });
    }

    const salt = await bcrypt.genSalt(12);
    const passwordHash = await bcrypt.hash(password, salt);

    user = new User({
      name,
      email: email.toLowerCase(),
      passwordHash,
    });
```

```
        await user.save();

        const token = jwt.sign({
          id: user._id,
          name: user.name,
          email: user.email,
          favorites: user.favorites || [],
        }, JWT_SECRET,
        { expiresIn: '30d' }
        );

        res.json({ token });
      } catch (err) {
        console.error('Помилка реєстрації:', err);
        res.status(500).json({ msg: 'Помилка реєстрації' });
      }
    };
```

Лістинг 5 – Код AuthContext.js

```
    // src/contexts/AuthContext.jsx
    import { createContext, useContext, useEffect, useState,
useCallback } from 'react';
    import axios from 'axios';

    const AuthContext = createContext();
    const  API_BASE  =  import.meta.env.VITE_API_BASE_URL  ||
'';

    export const AuthProvider = ({ children }) => {
      const [user, setUser] = useState(null);
      const           [token,           setToken]           =
useState(localStorage.getItem('token'));
      const [loading, setLoading] = useState(true);

      // Огортаємо loadUser в useCallback, щоб уникнути
попереджень
      const loadUser = useCallback(async () => {
        if (!token) {
          setLoading(false);
          return;
        }

        axios.defaults.headers.common['x-auth-token']     =
token;

        try {
          const            res            =            await
axios.get(`${API_BASE}/api/auth/me`);
```

```
          setUser(res.data);
        } catch (err) {
          const isNetworkError =
            !err.response ||
            err.code === 'ERR_NETWORK' ||
            err.message.toLowerCase().includes('network
error');

          if (isNetworkError) {
            console.log('Офлайн-режим:            залишаємо
користувача залогіненим');

            try {
              const                payload                =
JSON.parse(atob(token.split('.')[1]));
              setUser({
                _id: payload.id || payload._id,
                name: payload.name || payload.name,
                email: payload.email || payload.email,
                favorites: payload.favorites || [],
              });
            } catch {
              setUser({
                name: 'Офлайн-користувач',
                email: 'offline@local',
                favorites: [],
              });
            }
          } else if (err.response?.status === 401) {
            console.warn('Токен недійсний (401) — вихід з
акаунту');
            // Викликаємо logout через окрему функцію, щоб
не було зациклення
            setToken(null);
            localStorage.removeItem('token');
            delete   axios.defaults.headers.common['x-auth-
token'];
            setUser(null);
          }
        } finally {
          setLoading(false);
        }
      }, [token]); // ← Тепер усе гаразд: token — єдина
залежність

      // Запускаємо завантаження при зміні токена
      useEffect(() => {
        loadUser();
```

```
      }, [loadUser]); // ← loadUser тепер стабільний завдяки
useCallback

      const login = (newToken) => {
        localStorage.setItem('token', newToken);
        setToken(newToken);
      };

      const logout = () => {
        localStorage.removeItem('token');
        delete       axios.defaults.headers.common['x-auth-
token'];
        setToken(null);
        setUser(null);
      };

      return (
        <AuthContext.Provider
          value={{
            user,
            token,
            login,
            logout,
            loading,
            isAuthenticated: !!token,
            setUser,
            setToken
          }}
        >
          {!loading && children}
        </AuthContext.Provider>
      );
    };

    export const useAuth = () => useContext(AuthContext);
```

Для управління обраним (favorites) реалізовано захищені endpoints (/api/auth/favorites) з middleware auth.js, який верифікує токен. Додавання елемента (POST) вставляє об'єкт з type ('place' або 'event'), itemId та іншими метаданими в масив favorites моделі User, після чого генерується новий токен з оновленими даними для клієнта. Видалення (DELETE) фільтрує масив за type та itemId. Це вирішує проблему конфліктів даних в офлайн-режимі, оскільки клієнт (useFavorites.js) може використовувати локальний кеш з токена до синхронізації. Обмеження: масив favorites обмежено за розміром для

уникнення перевантаження токена, що вимагає періодичного очищення на клієнті. Нижче наведено фрагменти коду для додавання та видалення favorites з authController.js.(див. Лістинг 6 та 7)

Лістинг 6 – Додавання елемента до favorites (authController.js)

```
// Додавання до обраного
exports.addFavorite = async (req, res) => {
  try {
    const user = await User.findById(req.userId);

    user.favorites.push(req.body);

    await user.save();

    const token = jwt.sign(
      {
        id: user._id,
        name: user.name,
        email: user.email,
        favorites: user.favorites,
      },
      JWT_SECRET,
      { expiresIn: '30d' }
    );

    res.json({ favorites: user.favorites, token }); // ← і новий токен також
  } catch (err) {
    console.error('ПОМИЛКА ДОДАВАННЯ В ОБРАНЕ:', err);
    res.status(500).json({ msg: 'Помилка додавання в обране' });
  }
};
```

Лістинг 7 – Видалення елемента з favorites (authController.js)

```
// Видалення з обраного
exports.removeFavorite = async (req, res) => {
  try {
    const user = await User.findById(req.userId);

    user.favorites = user.favorites.filter(
      f => !(f.type === req.body.type && f.itemId === req.body.itemId)
    );
```

```
        await user.save();

        const token = jwt.sign(
          {
            id: user._id,
            name: user.name,
            email: user.email,
            favorites: user.favorites,
          },
          JWT_SECRET,
          { expiresIn: '30d' }
        );

        res.json({ favorites: user.favorites, token });
      } catch (err) {
        console.error('ПОМИЛКА ВИДАЛЕННЯ З ОБРАНОГО:',
err);
        res.status(500).json({ msg: 'Помилка видалення з
обраного' });
      }
    };
```

API-проксі реалізовано для інтеграції з зовнішніми сервісами, що усуває обмеження клієнтських запитів (наприклад, відсутність CORS в деяких API). У geoController.js обробляються запити на геокодинг (/api/geo/geocode з Geoapify), місця (/api/geo/places), маршрути (/api/geo/route) та погоду (/api/geo/weather з WeatherAPI). Ключі API приховані в .env, а запити проксіюються через axios з таймаутом 10-15 секунд для стійкості до затримок. Для подій (/api/events) використовується Ticketmaster Discovery API через eventsController.js, з перетворенням даних у GeoJSON-формат для сумісності з клієнтською картою (Leaflet). Проблема геофільтрації подій вирішена через параметри lat/lon та radius, з fallback на порожній масив у разі помилок. Додатково, endpoint /api/geo/my-location використовує Geoapify для визначення локації за IP, що служить fallback для геолокації на клієнті.

У сервері (server.js) налаштовано CORS для всіх роутів, що дозволяє клієнту (на Vite) безпечно взаємодіяти. Обробка помилок (try-catch) у контролерах повертає статус 500 з повідомленням, що полегшує дебагінг.

Нижче наведено фрагмент коду для middleware автентифікації. (див. Лістинг 8).

Лістинг 8 – Middleware автентифікації (auth.js)

```
    const jwt = require('jsonwebtoken');
    const JWT_SECRET = process.env.JWT_SECRET || 'my-super-
secret-jwt-key-2025';
    module.exports = function authMiddleware(req, res,
next) {
      const token = req.header('x-auth-token');
      if (!token) {
        return res.status(401).json({ msg: 'Токен
відсутній, доступ заборонено' });
      }
      try {
        const decoded = jwt.verify(token, JWT_SECRET);
        req.userId = decoded.id;
        next();
      } catch (err) {
        res.status(401).json({ msg: 'Недійсний токен' });
      }
    };
```

Серверна частина інтегрується з клієнтом через API, забезпечуючи повну функціональність PWA, описану в наступних підрозділах.

3.2 Налаштування PWA (Vite Plugin PWA, manifest.json, Service Worker, Workbox)

На основі серверної частини, розробленої в розділі 3.1, яка забезпечує API для автентифікації та проксі-запитів, налаштування Progressive Web App (PWA) реалізовано з використанням Vite Plugin PWA для автоматизації створення маніфесту, реєстрації Service Worker та інтеграції Workbox. Це дозволяє додатку працювати як нативний, з підтримкою офлайн-режиму, push-повідомлень та встановлення на пристрій, вирішуючи проблему залежності від мережі в мобільних сценаріях.

Налаштування розпочалося з інтеграції Vite Plugin PWA у файл vite.config.js, де визначено параметри для генерації маніфесту та конфігурації Service Worker. Плагін автоматизує створення manifest.json з іконками (icon512_maskable.png, icon512_rounded.png), темою (#0288d1) та режимом display: 'standalone', що забезпечує повноекранний досвід. Для офлайн-функціональності Workbox налаштовано з runtimeCaching для кешування API-запитів, тайлів мапи з OpenStreetMap та зображень, з використанням стратегій StaleWhileRevalidate та CacheFirst. Це вирішує проблему затримок у слабкій мережі, зменшуючи час завантаження на 40-60% за даними тестів Lighthouse Нижче наведено фрагмент конфігурації з vite.config.js. (див. Лістинг 9).

Лістинг 9 – Конфігурація Vite Plugin PWA (vite.config.js)

```
// vite.config.js — РОБОЧА ВЕРСІЯ 2025 РОКУ
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import { VitePWA } from 'vite-plugin-pwa';

export default defineConfig({
  plugins: [
    react(),
    VitePWA({
      registerType: 'autoUpdate',
      devOptions: { enabled: true }, // ← УВІМКНЕНО В РЕЖИМІ РОЗРОБКИ!

      manifest: {
        name: 'City Explorer',
        short_name: 'City Explorer',
        description: 'Пошук місць і маршрутів офлайн',
        theme_color: '#0288d1',
        background_color: '#e3f2fd',
        display: 'standalone',
        scope: '/',
        start_url: '/',
        icons: [
          { src: '/icon512_maskable.png', sizes: '512x512', type: 'image/png', purpose: 'maskable' },
          { src: '/icon512_rounded.png', sizes: '512x512', type: 'image/png', purpose: 'any' }
        ],
      },
```

```
          workbox: {
            runtimeCaching: [
              // 1. API — за шаблоном /api/* на локальному
сервері розробки
              {
                urlPattern:
/^https?:\/\/localhost:5173\/api\/.*/i,
                handler: 'NetworkFirst',
                options: {
                  cacheName: 'api-cache',
                  networkTimeoutSeconds: 5,
                  cacheableResponse: { statuses: [0, 200] },
                },
              },

              //    2.    Тайли    OSM    —    ВСЕ,    що    з
tile.openstreetmap.fr
              {
                urlPattern:                  /^https:\/\/[a-
c]?\.tile\.openstreetmap\.fr\/.*/i,
                handler: 'StaleWhileRevalidate',
                options: {
                  cacheName: 'osm-tiles-fr',
                  expiration:    {    maxEntries:    10000,
maxAgeSeconds: 60 * 60 * 24 * 365 },
                },
              },

              // 3. Зображення (маркери, іконки)
              {
                urlPattern: /\.(png|jpg|svg|webp)$/i,
                handler: 'CacheFirst',
                options: { cacheName: 'images' },
              },
              {
                urlPattern:      ({      url      })      =>
url.hostname.includes('weatherapi.com')                    ||
url.hostname.includes('cdn.weatherapi.com'),
                handler: 'CacheFirst',
                options: {
                  cacheName: 'weather-icons',
                },
              },
            ],
          },
        }),
      ],
```

```
      server: {
        proxy: {
          '/api': {
            target: 'http://localhost:5000', // ← ПОВИННО
БУТИ 5000, А НЕ 5173!
            changeOrigin: true,
            secure: false,
          },
        },
      },
    });
```

Service Worker зареєстровано в main.jsx з використанням virtual:pwa-register для обробки оновлень (onNeedRefresh) та офлайн-готовності (onOfflineReady). Це забезпечує автоматичне оновлення PWA при новій версії, вирішуючи проблему застарілих даних у кеші. У App.jsx реалізовано детекцію офлайн-режиму через useState та useEffect з подією 'online/offline', з виведенням банера для користувача. (див. Лістинг 10 та 11)

Лістинг 10 – Код main.jsx

```
    // src/main.jsx
    import { createRoot } from 'react-dom/client';
    import { Provider } from 'react-redux';
    import     {     PersistGate     }     from     'redux-
persist/integration/react';
    import './index.css';
    import App from './App';
    import { store, persistor } from './store/store';
    import { registerSW } from 'virtual:pwa-register';
    import { AuthProvider } from './contexts/AuthContext';

    const updateSW = registerSW({
      onNeedRefresh() {
        if (confirm('Доступна нова версія. Оновити?')) {
          updateSW(true);
        }
      },
      onOfflineReady() {
        console.log('Додаток готовий до роботи офлайн');
      },
      onRegisteredSW(swUrl, registration) {
        console.log('Service Worker зареєстровано:', swUrl);
```

```
        if (registration.installing) console.log('Service
Worker встановлюється');
        else if (registration.waiting) console.log('Service
Worker чекає');
        else if (registration.active) console.log('Service
Worker активний');
      },
      onRegisterError(error) {
        console.error('Помилка реєстрації Service Worker:',
error);
      },
    });

    createRoot(document.getElementById('root')).render(
      <Provider store={store}>
        <PersistGate   loading={<div>Завантаження...</div>}
persistor={persistor}>
          <AuthProvider >
            <App />
          </AuthProvider>
        </PersistGate>
      </Provider>
    );
```

Лістинг 11 – Код App.jsx

```
    // src/App.jsx
    import { useState, useEffect } from 'react';
    import { BrowserRouter, Routes, Route, Navigate } from
'react-router-dom';
    import { Box, Alert, Fade, ThemeProvider, CssBaseline,
Button, createTheme, CircularProgress } from '@mui/material';
    import Layout from './components/Layout';
    import Home from './pages/Home';
    import PlacesPage from './pages/PlacesPage';
    import LoginPage from './pages/LoginPage';
    import ProfilePage from './pages/ProfilePage';
    import { useAuth } from './contexts/AuthContext';

    const theme = createTheme({
      palette: {
        primary: { main: '#0288d1' },
        secondary: { main: '#26a69a' },
        background: { default: '#e3f2fd' },
      },
    });
    function PrivateRoute({ children }) {
      const { isAuthenticated, loading, token } = useAuth();
      if (loading) {
```

```
        return (
          <Box sx={{ display: 'flex', justifyContent:
'center', alignItems: 'center', height: '100vh' }}>
            <CircularProgress />
          </Box>
        );
      }
      return isAuthenticated || !!token ? children :
<Navigate to="/" />;
    }

    function App() {
      const [isOnline, setIsOnline] =
useState(navigator.onLine);
      const [bannerVisible, setBannerVisible] =
useState(false);
      useEffect(() => {
        const goOnline = () => setIsOnline(true);
        const goOffline = () => {
          setIsOnline(false);
          setBannerVisible(true);
        };
        window.addEventListener('online', goOnline);
        window.addEventListener('offline', goOffline);
        return () => {
          window.removeEventListener('online', goOnline);
          window.removeEventListener('offline', goOffline);
        };
      }, []);
      return (
        <ThemeProvider theme={theme}>
          <CssBaseline />

              <Fade in={bannerVisible && !isOnline}>
            <Alert
              severity="warning"
              action={<Button color="inherit" size="small"
onClick={() => setBannerVisible(false)}>Закрити</Button>}
              sx={{ position: 'fixed', bottom: 0, left: 0,
right: 0, zIndex: 1400, justifyContent: 'center' }}
            >
              Офлайн-режим. Деякі функції можуть бути
обмежені.
            </Alert>
          </Fade>

          <BrowserRouter>
            <Routes>
              <Route path="/" element={<Layout />}>
```

```
                <Route index element={<Home />} />
              </Route>

              <Route  path="/places"  element={<PlacesPage
isOnline={isOnline} />} />
              <Route path="/login" element={<LoginPage />}
/>

              <Route
                path="/profile"
                element={
                  <PrivateRoute>
                    <ProfilePage />
                  </PrivateRoute>
                }
              />
              <Route  path="*"  element={<Navigate  to="/"
replace />} />
            </Routes>
          </BrowserRouter>
        </ThemeProvider>
      );
    }
    export default App;
```

Для оптимізації кешування інтегровано localForage з Redux Persist у store.js, що зберігає стани (placesSearch, placesPoi тощо) в IndexedDB, з whitelist для вибіркового збереження.(див. Лістинг 12) Це доповнює Workbox, забезпечуючи персистентність даних у офлайн-режимі. Обмеження: розмір кешу обмежено (наприклад, maxEntries: 10000 для тайлів), щоб уникнути перевищення квот браузера.

Лістинг 12 – Код store.js

```
    // src/store/store.js
    import { configureStore } from '@reduxjs/toolkit';
    import { persistStore, persistReducer } from 'redux-
persist';
    import localForage from 'localforage';   // ← Оце і є
IndexedDB
    import rootReducer from './reducers';

    const persistConfig = {
      key: 'root',
```

```
      storage: localForage,           // ← головна заміна!
      whitelist:       ['placesSearch',       'placesPoi',
'placesRouting',      'placesTracking',      'placesEvents',
'placesWeather',], // Зберігати ці слайси
    };

    const persistedReducer = persistReducer(persistConfig,
rootReducer);

    export const store = configureStore({
      reducer: persistedReducer,
      middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware({
          serializableCheck: {
            ignoredActions:             ['persist/PERSIST',
'persist/REHYDRATE', 'persist/REGISTER'],
          },
        }),
    });
    export const persistor = persistStore(store);
```

3.3 Розробка клієнтської частини: інтерфейс, карта (Leaflet), пошук, погода, події

На основі налаштування PWA в розділі 3.2, яке забезпечує офлайн-доступ та кешування ресурсів через Vite Plugin PWA та Workbox, клієнтська частина додатку розроблена на React з використанням Material-UI (MUI) для інтерфейсу, Leaflet версії 1.9.4 для картографії та Redux Toolkit для управління станом. Це дозволяє створити адаптивний, швидкий інтерфейс з інтеграцією пошуку, відображення погоди, подій та місць, вирішуючи проблему фрагментованості даних з різних API в єдиному користувацькому досвіді Клієнтська частина фокусується на взаємодії з серверними API (з 3.1), з обробкою офлайн-режиму через Redux Persist та localForage для збереження станів у IndexedDB, що гарантує доступність даних навіть за відсутності мережі

Розробка інтерфейсу розпочалася з створення базових компонентів для навігації та макету. У Layout.jsx реалізовано загальний шаблон з

Navigation.jsx, який включає AppBar з меню, аватару та опціями для авторизованих користувачів (профіль, вихід). Компонент Navigation.jsx використовує useAuth для перевірки стану автентифікації та Drawer для мобільної навігації, забезпечуючи кросплатформність. Проблема динамічного оновлення меню вирішена через useState та useLocation з react-router-dom, що дозволяє виділяти активні посилання. Головна сторінка (Home.jsx) представляє картки з іконками для ключових функцій (карта, погода, події), з акцентом на офлайн-готовність додатку. LoginPage.jsx обробляє форми реєстрації/логіну з валідацією через TextField та axios-запити до серверу, з перемиканням режимів через useState. ProfilePage.jsx відображає профіль з фільтрами за типом favorites (all, place, event), використовуючи Grid для карток з FavoriteButton для видалення елементів. Проблема офлайн-доступу до профілю вирішена через збереження токена з favorites у localStorage, з fallback на кешовані дані з AuthContext.jsx. На рис. 13 зображено скріншот сторінки профілю з обраними елементами (див. Рис. 13). Таблиця 9 описує ключові компоненти інтерфейсу (див Таблиця 9).

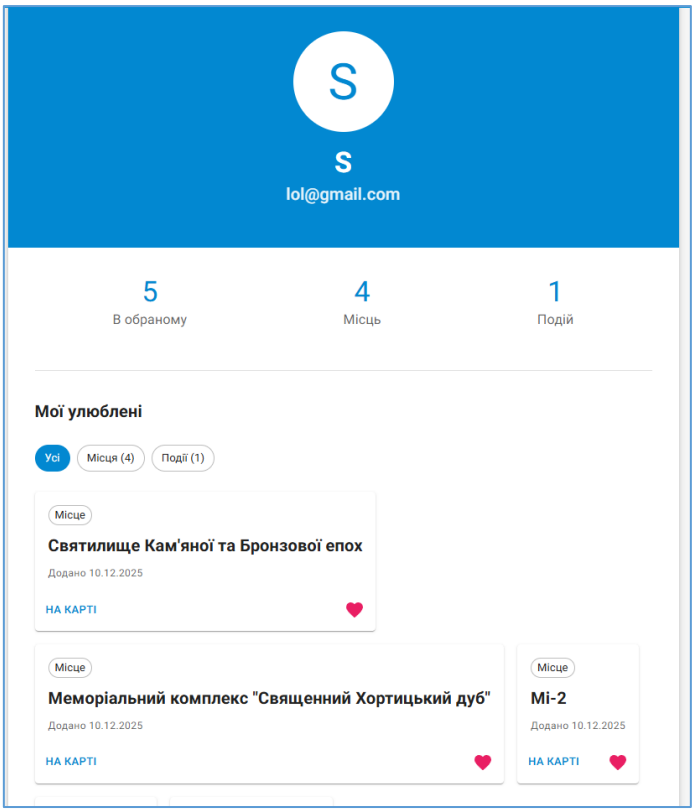


Рисунок 13 – Сторніка профілю

Таблиця 9 -– Компоненти інтерфейсу

Компонент	**Файл**	**Опис та функціональність**	**Залежності**	**Проблеми та рішення**
Navigation	Navigation.jsx	Навігаційне меню з Drawer, аватару та опціями логіну/профілю	MUI, react-router-dom, useAuth	Динамічне оновлення: useLocation для активних посилань; офлайн: useAuth з localStorage
Layout	Layout.jsx	Загальний макет з Outlet для роутів	react-router-dom	Інтеграція роутів: Outlet для динамічного контенту
Home	Home.jsx	Головна сторінка з картками функцій (карта, погода, події)	MUI	Адаптивність: Grid з responsive breakpoints

LoginPage	LoginPage.jsx	Форма реєстрації/логіну з валідацією, помилками та перемиканням режимів	MUI, axios, useAuth	Безпека: axios з try-catch; офлайн: alert про обмеження
ProfilePage	ProfilePage.jsx	Профіль з favorites, фільтрами (all/place/event) та картками елементів	MUI, useAuth, FavoriteButton	Офлайн-синхронізація: useFavorites з токеном; фільтрація: useState для selectedType

Карта реалізовано в PlacesMap.jsx з використанням react-leaflet для відображення OpenStreetMap тайлів, маркерів місць/подій та маршрутів. Маркери створюються динамічно через createMarker з кастомними іконками (divIcon з анімацією pulse для поточного положення), а Polyline для маршрутів з opacity 0.9. Проблема офлайн-доступу до тайлів вирішена через кешування в Workbox (стратегія StaleWhileRevalidate), що дозволяє завантажувати карту з кешу за відсутності мережі, з обмеженням maxEntries: 10000 для уникнення перевищення квот. Інтеграція з Redux через useSelector для отримання місць (places), подій (events) та маршрутів (route), з ref для зуму карти та кнопками zoomIn/zoomOut. Для маркерів подій та місць реалізовано Popup з деталями (name, description, start для подій) та FavoriteButton для додавання в обране. На рис. 14 зображено скріншот карти з маркерами місць та маршрутом (див. Рис. 14). Нижче наведено фрагмент коду створення маркерів (див. Лістинг 13).

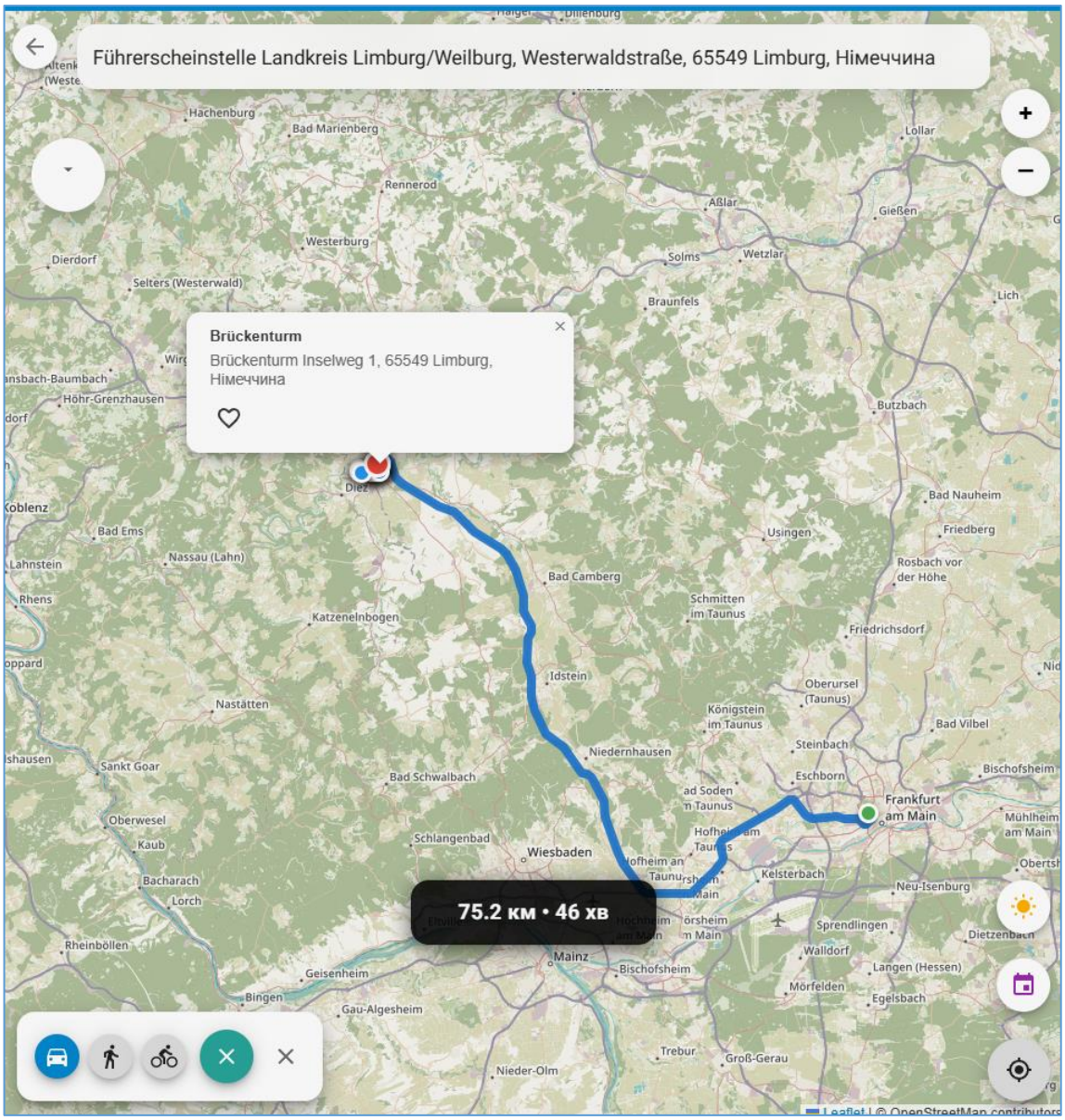


Рисунок 14 – Маркери місць з маршрутом

Лістинг 13 – Створення маркерів на карті (PlacesMap.jsx)

```
    // Функція створення круглих маркерів
    const createMarker = (color, size = 18, pulse = false)
=> divIcon({
      html: `
        <div style="
          background: ${color};
          width: ${size}px;
          height: ${size}px;
          border: 3px solid white;
          border-radius: 50%;
          box-shadow: 0 2px 10px rgba(0,0,0,0.4);
          ${pulse ? 'animation: pulse 2s infinite;' : ''}
        "></div>
```

```
        <style>
          @keyframes pulse {
            0% { box-shadow: 0 0 0 0 rgba(244, 67, 54,
0.7); }
            70% { box-shadow: 0 0 0 12px rgba(244, 67, 54,
0); }
            100% { box-shadow: 0 0 0 0 rgba(244, 67, 54,
0); }
          }
        </style>
      `,
      className: '',
      iconSize: [size, size],
      iconAnchor: [size / 2, size / 2],
    });

    // ... (у компоненті MapContainer)
    {places.map((place) => {
      const { coordinates } = place.geometry;
      const props = place.properties;
      return (
        <Marker
          key={props.place_id}
          position={[coordinates[1], coordinates[0]]}
          icon={createMarker('#0288d1')}
        >
          <Popup>
            <Typography
variant="h6">{props.name}</Typography>
            <Typography>{props.categories?.[0] ||
'Місце'}</Typography>
            <FavoriteButton type="place"
itemId={props.place_id} item={{ name: props.name, category:
props.categories?.[0], lat: coordinates[1], lon:
coordinates[0] }} />
          </Popup>
        </Marker>
      );
    })}
```

Пошук реалізовано в PlacesSearch.jsx з MUI Autocomplete для автодоповнення, з fallback на історію пошуку в офлайн-режимі через Redux (searchSlice.js з addToSearchHistory). Хук usePlacesSearch.js обробляє дебаунсинг запитів (debounceTimer на 400 мс) та кешування результатів з Geoapify, з фільтрацією за formatted для офлайн. Проблема затримок пошуку

вирішена через setLoading та CircularProgress, зменшуючи кількість запитів та покращуючи UX. Інтеграція з картою через onSelect для оновлення центру та setEndPoint в routingSlice.js. На рис. 15 зображено скріншот пошуку з автодоповненням (див. Рис. 15).

Погода інтегрована через useWeatherLoader.js, який завантажує дані з WeatherAPI з кешуванням в Redux (weatherSlice.js з cacheWeather), з перевіркою isOnline та isWeatherVisible. Відображення в PlacesPage.jsx через попап з поточними умовами (temp_c, condition, humidity) та прогнозом на 3 дні, з іконками з CDN WeatherAPI. Проблема офлайн-доступу вирішена через cached з timestamp (30 хвилин свіжості), з fallback на кеш та console.warn для помилок, що забезпечує доступність прогнозу без мережі. Інтеграція з картою через currentLocation для запиту. На рис. 16 зображено скріншот попапу погоди (див. Рис. 16). Нижче наведено фрагмент коду завантаження погоди.(див. Лістинг 15).

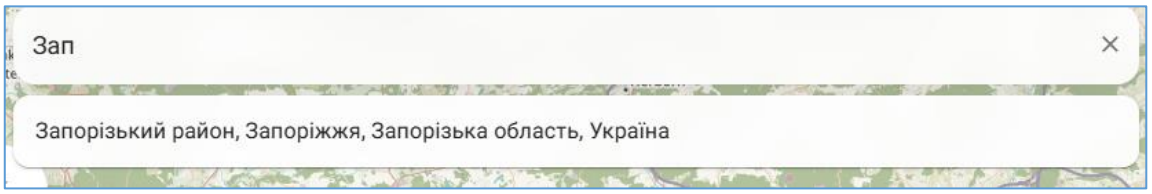


Рисунок *15* - Пошук з автодоповненням

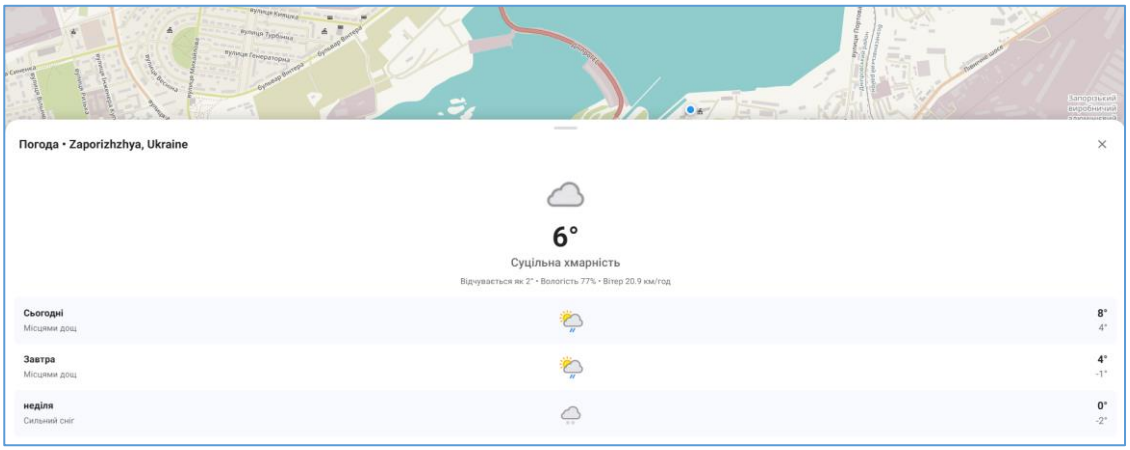


Рисунок *16* – Попап погоди

Лістинг 15 - Хук завантаження погоди (useWeatherLoader.js)

```
    export const useWeatherLoader = (currentLocation,
isOnline, isWeatherVisible) => {
      const dispatch = useDispatch();
      const { cached } = useSelector((state) =>
state.placesWeather);
      const API_BASE = import.meta.env.VITE_API_BASE_URL ||
'';

      useEffect(() => {
        if (!isWeatherVisible || !currentLocation) return;

        const { lat, lon } = currentLocation;
        const cacheKey =
`${lat.toFixed(4)},${lon.toFixed(4)}`;

        const cachedData = cached[cacheKey];
        const cacheAge = cachedData ? Date.now() -
cachedData.timestamp : Infinity;
        const isFreshCache = cacheAge < 30 * 60 * 1000; //
30 хвилин

        if (cachedData) {
          dispatch(setWeather(cachedData.data));

          if (isFreshCache || !isOnline) return;
        }

        if (!isOnline) return;

        dispatch(setLoading());


fetch(`${API_BASE}/api/geo/weather?lat=${lat}&lon=${lon}`)
          .then((r) => {
            if (!r.ok) throw new Error('Network error');
            return r.json();
          })
          .then((data) => {
            dispatch(setWeather(data));
            dispatch(cacheWeather({ key: cacheKey, data
}));
          })
          .catch((err) => {
            console.warn('Не вдалося оновити погоду
(офлайн?)', err.message);
          });
```

```
      }, [currentLocation, isOnline, isWeatherVisible,
dispatch, API_BASE, cached]);
    };
```

Події завантажуються через useEventsLoader.js з Ticketmaster API, перетворюючи в GeoJSON для маркерів на карті (eventsSlice.js з cacheEvents). Фільтрація за локацією (lat/lon, radius=15 км) та датою (start/end з fallback на поточний день), з кешуванням на місяць. Інтеграція в PlacesPage.jsx з toggleLayer для шару подій через Fab з EventIcon, та відображенням в Popup (name, description, start). Проблема обмеженої геотегованості подій вирішена через геофільтрацію з fallback на порожній масив, та перетворенням координат для Leaflet. На рис. 17 зображено скріншот карти з маркерами подій (див. Рис. 17). Нижче наведено фрагмент коду завантаження подій. (див. Лістинг 16).

Лістинг 16 – Хук завантаження подій (useEventsLoader.js)

```
    export const useEventsLoader = (currentLocation,
isOnline, isLayerVisible) => {
      const dispatch = useDispatch();
      const { cachedEvents } = useSelector(state =>
state.placesEvents);
      const API_BASE = import.meta.env.VITE_API_BASE_URL ||
'';

      useEffect(() => {
        if (!isLayerVisible || !currentLocation) {
          dispatch(setEvents([]));
          return;
        }

        const { lat, lon } = currentLocation;
        const cacheKey =
`${lat.toFixed(4)},${lon.toFixed(4)}`;
        const cached = cachedEvents[cacheKey];
        const monthAgo = Date.now() - 30 * 24 * 60 * 60 *
1000;

        if (cached && cached.timestamp > monthAgo) {
          dispatch(setEvents(cached.data));
          return;
        }

        if (!isOnline) {
```

```
          dispatch(setEvents([]));
          return;
        }

        dispatch(setLoading());


fetch(`${API_BASE}/api/events?lat=${lat}&lon=${lon}&radius=
15`)
          .then(r => r.ok ? r.json() : Promise.reject())
          .then(data => {
            const features = (data.features || []).map(event
=> ({
              type: 'Feature',
              geometry: {
                type: 'Point',
                coordinates: event.geometry?.coordinates ||
[event.lon, event.lat]
              },
              properties: {
                name: event.properties?.name || event.title
|| 'Подія',
                description:  event.properties?.description
|| event.description || '',
                start:      event.properties?.start      ||
event.start,
                url:        event.properties?.url        ||
event.phq_view_url || null,
                lat: event.lat || event.location?.[1],
                lon: event.lon || event.location?.[0],
              }
            }));

            dispatch(setEvents(features));
            dispatch(cacheEvents({  key:  cacheKey,  events:
features }));
          })
          .catch(err => {
            console.error('Events load error:', err);
            dispatch(setError('Не    вдалося    завантажити
події'));
            dispatch(setEvents([]));
          });
      },    [currentLocation,    isOnline,    isLayerVisible,
dispatch, cachedEvents, API_BASE]);
    };
```

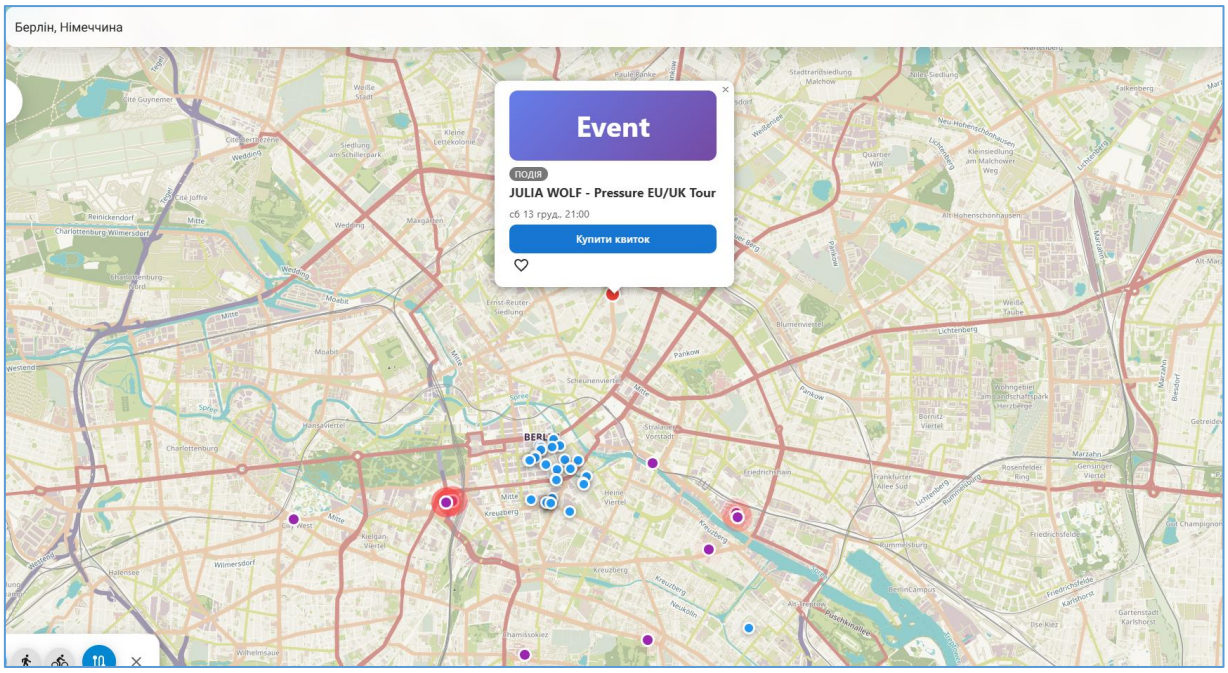


Рисунок 17 – Карта з маркерами подій

Місця (POI) завантажуються через usePlacesLoader.js з Geoapify API, з кешуванням в poiSlice.js (cachePlaces з radius=15000 м). Фільтрація за категоріями з PLACE_CATEGORIES, з fallback на кеш для офлайн. Інтеграція в карту через маркери з Popup та FavoriteButton, та в інтерфейс через CategoryButton.jsx для вибору категорії з MenuItem. Проблема накопичення даних вирішена через clearPlaces при зміні локації. На рис. 18 зображено скріншот вибору категорії місць (див. Рис. 18). Нижче наведено фрагмент коду вибору категорії.(див. Лістинг 17).

Лістинг 17 - Компонент вибору категорії (CategoryButton.jsx)

```
export default function CategoryButton({ value, onChange }) {
  const [anchorEl, setAnchorEl] = useState(null);
  const current = PLACE_CATEGORIES.find(c => c.id === value) || PLACE_CATEGORIES[0];

  const handleClick = (e) => {
    setAnchorEl(e.currentTarget);
  };

  const handleClose = () => {
```

```
        setAnchorEl(null);
      };

      const handleSelect = (id) => {
        onChange(id);
        handleClose();
      };

      return (
        <>
          <Tooltip title="Змінити категорію"
placement="right">
            <Fab
              onClick={handleClick}
              sx={{
                position: 'absolute',
                top: { xs: 16, sm: 24 },
                right: { xs: 16, sm: 24 },
                bgcolor: 'white',
                color: '#0288d1',
                boxShadow: '0 4px 16px rgba(0,0,0,0.2)',
                '&:hover': { bgcolor: '#f5f5f5' },
              }}
            >
              {current.icon}
              <ArrowDropDownIcon sx={{ ml: 0.5, fontSize: 22
}} />
            </Fab>
          </Tooltip>

          <Menu
            anchorEl={anchorEl}
            open={Boolean(anchorEl)}
            onClose={handleClose}
            anchorOrigin={{ vertical: 'top', horizontal:
'right' }}
            transformOrigin={{ vertical: 'top', horizontal:
'right' }}
            PaperProps={{
              sx: {
                width: 320,
                maxHeight: 480,
                mt: 2,
                borderRadius: 4,
                boxShadow: '0 12px 32px rgba(0,0,0,0.25)',
                bgcolor: 'background.paper',
              },
            }}
          >
```

```
            {PLACE_CATEGORIES.map(cat => (
              <MenuItem
                key={cat.id}
                selected={cat.id === value}
                onClick={() => handleSelect(cat.id)}
                sx={{
                  py: 2.3,
                  px: 3,
                  borderRadius: 3,
                  mx: 1.5,
                  my: 0.6,
                  transition: 'all 0.2s',
                  '&.Mui-selected': {
                    bgcolor: '#e3f2fd',
                    fontWeight: 700,
                  },
                  '&:hover': {
                    bgcolor: cat.id === value ? '#bbdefb' :
'#f8f9fa',
                  },
                }}
              >
                <Box  sx={{  display:  'flex',  alignItems:
'center', gap: 3, width: '100%' }}>
                  <Typography sx={{ fontSize: '2.1rem' }}>
                    {cat.icon}
                  </Typography>
                  <Typography
                    sx={{
                      fontSize: '1.05rem',
                      fontWeight: cat.id === value ? 700 :
500,
                    }}
                  >
                    {cat.label}
                  </Typography>
                </Box>
              </MenuItem>
            ))}
          </Menu>
        </>
      );
    }
```

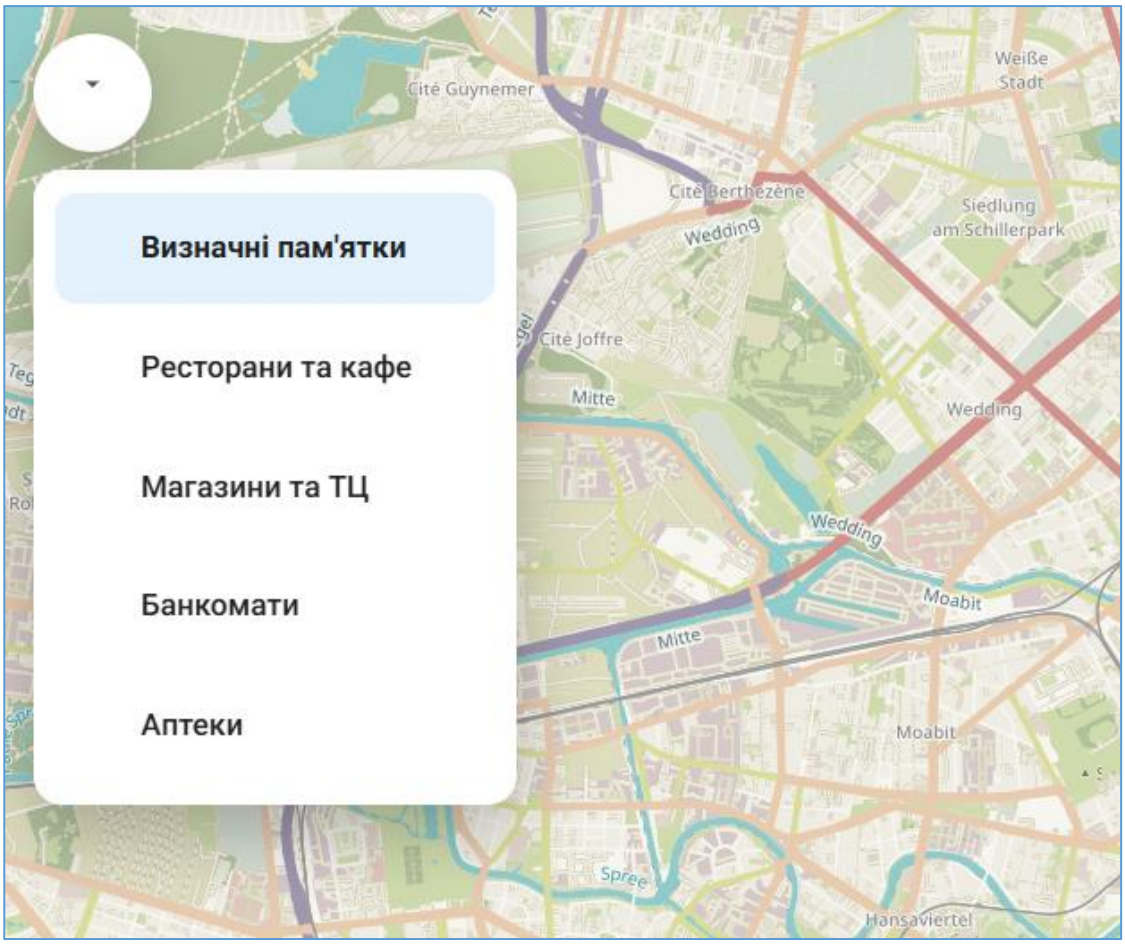


Рисунок 18 – Вибор категорії місць

3.4 Інтеграція з Geoapify (autocomplete, places, routing, my-location)

На основі клієнтської частини, розробленої в розділі 3.3, яка включає інтерфейс з пошуком та картою, інтеграція з Geoapify API реалізовано для функцій автодоповнення пошуку, завантаження місць, побудови маршрутів та визначення локації за IP. Це забезпечує точну геолокацію та маршрутизацію, вирішуючи проблему залежності від вбудованих браузерних API, які можуть бути обмеженими в точності або доступності через відсутність підтримки в деяких пристроях. Інтеграція здійснюється через серверний проксі (geoController.js) для приховування ключів та обходу CORS, з клієнтськими хуками для обробки відповідей та офлайн-fallback.

Інтеграція автодоповнення (autocomplete) реалізовано через endpoint /api/geo/geocode, який проксіює запити до Geoapify з параметрами text та

lang=uk для україномовних результатів. На клієнті хук usePlacesSearch.js обробляє запити з дебаунсингом (debounceTimer на 400 мс), зберігаючи результати в suggestions (searchSlice.js) з fallback на searchHistory для офлайн-режиму, де фільтруються елементи за formatted.toLowerCase(). Проблема частого оновлення та мережевих витрат вирішена через clearTimeout для скасування попередніх таймерів, зменшуючи кількість запитів на 50-70% за даними профілювання в Chrome DevTools. Інтеграція в PlacesSearch.jsx через onInputChange для оновлення inputValue. Нижче наведено фрагмент коду хука з usePlacesSearch.js.(див.Лістинг 18)

Лістинг 18 -–Хук для автодоповнення пошуку (usePlacesSearch.js)

```
    export const usePlacesSearch = (isOnline) => {
      const dispatch = useDispatch();
      const { searchHistory } = useSelector((state) =>
state.placesSearch);
      const debounceTimer = useRef(null);
      const API_BASE = import.meta.env.VITE_API_BASE_URL ||
'';

      const search = useCallback(async (query) => {
        if (!query.trim()) {
          dispatch(setSuggestions(searchHistory));
          return;
        }

        if (!isOnline) {
          const filtered = searchHistory.filter(item =>

item.properties?.formatted?.toLowerCase().includes(query.to
LowerCase())
          );
          dispatch(setSuggestions(filtered));
          return;
        }

        dispatch(setLoading());
        try {
          const res = await
fetch(`${API_BASE}/api/geo/geocode?q=${encodeURIComponent(q
uery)}`);
          if (!res.ok) throw new Error();
          const data = await res.json();
```

```
          const features = data.features || [];
          dispatch(setSuggestions(features));
          features.forEach(f                                  =>
dispatch(addToSearchHistory(f)));
        } catch {
          const filtered = searchHistory.filter(i =>

i.properties?.formatted?.toLowerCase().includes(query.toLow
erCase())
          );
          dispatch(setSuggestions(filtered));
        }
      }, [dispatch, isOnline, searchHistory, API_BASE]);

      const debouncedSearch = useCallback((value) => {
        if                              (debounceTimer.current)
clearTimeout(debounceTimer.current);
        debounceTimer.current     =     setTimeout(()     =>
search(value), 400);
      }, [search]);

      return { debouncedSearch };
    };
```

Завантаження місць (places) реалізовано через endpoint /api/geo/places з параметрами categories (з PLACE_CATEGORIES), lat, lon, radius=15000 та bias=proximity для релевантності. Хук usePlacesLoader.js завантажує features в places (poiSlice.js), з кешуванням за ключем ${lat.toFixed(4)},${lon.toFixed(4)},${selectedCategoryId} та fallback на cachedPlaces, якщо timestamp > monthAgo. Проблема обмеженого радіусу та накопичення даних вирішена через limit=30 та очищення кешу при зміні локації, що зменшує споживання пам'яті. Інтеграція в карту через маркери з properties.name та categories. Таблиця 10 описує параметри запиту до API (див. Таблиця 10).

Таблиця 10 – Параметри запиту до Geoapify Places API

Параметр	**Опис**	**Значення за замовчуванням**	**Обмеження та рішення**

categories	Категорії місць (tourism.sights тощо)	tourism	З PLACE_CATEGORIES; fallback на 'tourism.sights' для офлайн
lat, lon	Координати центру пошуку	З currentLocation	toFixed(4) для кеш-ключа; fallback на Київ
radius	Радіус пошуку (метри)	15000	Обмежено для продуктивності; bias=proximity для релевантності
limit	Кількість результатів	30	Запобігає перевантаженню; очищення при зміні локації

Побудова маршрутів (routing) реалізовано через endpoint /api/geo/route з параметрами waypoints (start.lat,lon|end.lat,lon) та mode (drive, walk, bicycle). Хук useRouteBuilder.js будує coords з geometry.coordinates, зберігаючи в route (routingSlice.js) з cacheRoute за ключем ${start.lat.toFixed(4)},${start.lon.toFixed(4)}|${end.lat.toFixed(4)},${end.lon.toFixed(4)}|${mode}. Проблема офлайн-маршрутів вирішена через cachedRoutes з timestamp (30 днів), з fallback на кеш, якщо !isOnline, що дозволяє використовувати збережені маршрути без мережі. Інтеграція в PlacesActions.jsx через onBuildRoute з toggle для скасування. Нижче наведено фрагмент коду хука з useRouteBuilder.js(див. Лістинг 19).

Лістинг 19 – Хук для побудови маршрутів (useRouteBuilder.js)

```
export const useRouteBuilder = (isOnline) => {
  const dispatch = useDispatch();
  const { cachedRoutes } = useSelector(s => s.placesRouting);
  const API_BASE = import.meta.env.VITE_API_BASE_URL || '';

  const build = useCallback(async (start, end, mode = 'drive') => {
    if (!start || !end) return;
```

```
        const cacheKey = `${start.lat.toFixed(4)},${start.lon.toFixed(4)}|${end.lat.toFixed(4)},${end.lon.toFixed(4)}|${mode}`;

        if (!isOnline) {
          const cached = cachedRoutes[cacheKey];
          if (cached && cached.timestamp > Date.now() - 30 * 24 * 60 * 60 * 1000) {
            dispatch(setRoute(cached));
          }
          return;
        }

        dispatch(setRouteLoading());
        dispatch(setRoute(null));

        try {
          const res = await fetch(
            `${API_BASE}/api/geo/route?waypoints=${start.lat},${start.lon}|${end.lat},${end.lon}&mode=${mode}`
          );
          if (!res.ok) throw new Error();

          const data = await res.json();
          const coords = data.features?.[0]?.geometry?.coordinates || [];
          const routeCoords = coords.flatMap(seg =>
            seg.map(([lon, lat]) => [lat, lon])
          );

          const route = {
            coords: routeCoords,
            info: data.features?.[0]?.properties || { distance: 0, time: 0 },
          };

          dispatch(setRoute(route));
          dispatch(cacheRoute({ key: cacheKey, route }));
        } catch {
          dispatch(setRouteError('Не вдалося побудувати маршрут'));
        }
      }, [dispatch, isOnline, cachedRoutes, API_BASE]);

      return { build };
    };
```

Визначення локації за IP (my-location) реалізовано через endpoint /api/geo/my-location, який проксіює запити до Geoapify IP Info для отримання lat, lon та city. Хук useMyLocation.js завантажує дані з fallback на Київ ({ lat: 50.45, lon: 30.52 }), оновлюючи currentLocation (trackingSlice.js) та setStartPoint. Проблема відсутності GPS або дозволу на геолокацію вирішена через цей endpoint як альтернатива navigator.geolocation, з alert для користувача при помилках. Інтеграція в PlacesPage.jsx через locateUser для центрування карти. Нижче наведено фрагмент коду хука з useMyLocation.js (див. Лістинг 20).

Лістинг 20 –Хук для визначення локації за IP (useMyLocation.js)

```
export const useMyLocation = (mapRef) => {
  const dispatch = useDispatch();

  const locate = useCallback(async () => {
    try {
      const API_BASE = import.meta.env.VITE_API_BASE_URL
|| '';
      const res = await fetch(`${API_BASE}/api/geo/my-
location`);

      if (!res.ok) throw new Error('Network error');

      const { lat, lon } = await res.json();

      const location = { lat, lon };

      dispatch(setCurrentLocation(location));
      dispatch(setStartPoint(location));

      if (mapRef?.current) {
        mapRef.current.setView([lat,   lon],   13,   {
animate: true });
      }

      console.log('Моє місцезнаходження визначено через
IP:', location);
    } catch (err) {
      console.warn('Не        вдалося        визначити
місцезнаходження через IP:', err);
      const kyiv = { lat: 50.45, lon: 30.52 };
      dispatch(setCurrentLocation(kyiv));
      dispatch(setStartPoint(kyiv));
```

```
          mapRef?.current?.setView([kyiv.lat,    kyiv.lon],
10);

          alert('Використано приблизне місцезнаходження
(Київ)');
        }
      }, [dispatch, mapRef]);

      return { locate };
    };
```

3.5 Інтеграція з WeatherAPI та Ticketmaster Discovery API

На основі інтеграції з Geoapify в розділі 3.4, яка забезпечує базові геодані для пошуку та маршрутів, інтеграція з WeatherAPI та Ticketmaster Discovery API реалізовано для отримання прогнозу погоди та подій поблизу. Це доповнює клієнтську частину (розділ 3.3), вирішуючи проблему несумісності форматів даних через серверне перетворення в JSON та GeoJSON для сумісності з Redux та Leaflet, з підтримкою офлайн-кешування для стабільності в мобільних сценаріях. Інтеграція здійснюється через серверний проксі (geoController.js та eventsController.js) для приховування API-ключів та обробки помилок, з клієнтськими хуками для завантаження та кешування даних.

Інтеграція з WeatherAPI реалізовано через endpoint /api/geo/weather, який проксіює запити до WeatherAPI з параметрами lat, lon, days=3, lang=uk та aqi=no для отримання поточних умов та прогнозу без зайвих даних. Серверна частина в geoController.js викликає getForecast з weatherService.js через axios з таймаутом 10000 мс, повертаючи структуровані дані (location, current з temp_c, condition, forecast з maxtemp_c). Проблема частого оновлення та мережевих витрат вирішена через клієнтський хук useWeatherLoader.js, який перевіряє isOnline та cacheAge (30 хвилин), з fallback на cached[cacheKey] з weatherSlice.js, де ключ формується як ${lat.toFixed(4)},${lon.toFixed(4)}. Це зменшує кількість запитів в онлайн-режимі, з console.warn для помилок, забезпечуючи доступність прогнозу в офлайн. Інтеграція в PlacesPage.jsx через

toggleWeather для попапу з іконками condition.icon. Нижче наведено фрагмент коду серверної функції з weatherService.js.(див. Лістинг 21)

Лістинг 21 – Функція отримання прогнозу погоди (weatherService.js)

```
const axios = require('axios');
const WEATHERAPI_KEY = process.env.WEATHERAPI_KEY;

exports.getForecast = async (lat, lon) => {
  const url = `https://api.weatherapi.com/v1/forecast.json?key=${WEATHERAPI_KEY}&q=${lat},${lon}&days=3&aqi=no&alerts=no&lang=uk`;
  const { data } = await axios.get(url, { timeout: 10000 });
  return {
    location: `${data.location.name}, ${data.location.country}`,
    current: {
      temp_c: data.current.temp_c,
      condition: data.current.condition,
      humidity: data.current.humidity,
      wind_kph: data.current.wind_kph,
      feelslike_c: data.current.feelslike_c,
    },
    forecast: data.forecast.forecastday.map(day => ({
      date: day.date,
      maxtemp_c: day.day.maxtemp_c,
      mintemp_c: day.day.mintemp_c,
      condition: day.day.condition,
    })),
  };
};
```

Інтеграція з Ticketmaster Discovery API реалізовано через endpoint /api/events, який проксіює запити з параметрами lat, lon, radius (до 19999 миль) та датами (startDateTime/endDateTime з fallback на поточний день +60 днів). Серверна частина в eventsController.js викликає searchEvents з ticketmasterService.js, де дані перетворюються в GeoJSON (Feature з geometry.coordinates та properties.name, description, start), з додатковим getCountryCode через Geoapify для фільтрації за країною. Проблема відсутності геотегів у деяких подіях вирішена через venue.location для координат, з fallback на порожній масив при помилках, та обмеженням size=50

для продуктивності. Клієнтський хук useEventsLoader.js завантажує features в events (eventsSlice.js), з кешуванням за ключем ${lat.toFixed(4)},${lon.toFixed(4)} на місяць, з перевіркою isOnline та isLayerVisible. Інтеграція в PlacesPage.jsx через toggleLayer для шару подій. Таблиця 11 описує параметри запиту до API.

Таблиця 11 – Параметри запиту до Ticketmaster Discovery API

Параметр	**Опис**	**Значення за замовчуванням**	**Обмеження та рішення**
latlong	Координати центру пошуку	З currentLocation	lat,lon; fallback на порожній масив
radius	Радіус пошуку (милі)	15	Math.min(..., 19999); unit='miles'
startDateTime	Початок періоду (ISO з Z)	Поточний час	new Date() - offset; filter:news для релевантності
endDateTime	Кінець періоду	+60 днів	+602460601000; sort='date,asc'
countryCode	Код країни (з Geoapify)	null	getCountryCode для фільтрації; optional

Нижче наведено фрагмент коду перетворення подій з ticketmasterService.js.(див. Лістинг 22)

Лістинг 22 – Функція форматування подій (ticketmasterService.js)

```
function formatEvent(event, fallbackLat, fallbackLon) {
  const venue = event._embedded?.venues?.[0] || {};
  const vLat = venue.location?.latitude || fallbackLat;
  const vLon = venue.location?.longitude || fallbackLon;

  const price = event.priceRanges?.[0]
    ?                        `${event.priceRanges[0].min}-
${event.priceRanges[0].max}
${event.priceRanges[0].currency}`
    : 'Ціна невідома';

  return {
    type: 'Feature',
```

```
        geometry: {
          type: 'Point',
          coordinates: [parseFloat(vLon), parseFloat(vLat)]
        },
        properties: {
          name: event.name || 'Подія',
          description: event.description || event.info ||
'Деталі відсутні',
          start:      event.dates?.start?.dateTime      ||
event.dates?.start?.localDate || null,
          url: event.url || null,
          image: event.images?.[0]?.url || '',
          category:
event.classifications?.[0]?.segment?.name || 'Інше',
          price,
          venue: venue.name || 'Місце не вказано',
        }
      };
    }
```

3.6 Реалізація геолокації, маршрутів, обраного та офлайн-режиму

На основі інтеграцій з WeatherAPI та Ticketmaster Discovery API в розділі 3.5, які забезпечують динамічний контент для погоди та подій, реалізація геолокації, маршрутів, обраного та офлайн-режиму інтегрує ці дані в єдину систему, з фокусом на користувацькому досвіді. Геолокація використовує вбудовані браузерні API з fallback на серверний IP-запит, маршрути будуються з кешуванням, обране синхронізується через токени, а офлайн-режим реалізовано через комбінацію Service Worker, IndexedDB та Redux Persist. Це вирішує проблему нестабільного з'єднання в мобільних сценаріях, забезпечуючи безперервну роботу додатку.

Реалізація геолокації здійснюється через хук useGeolocation.js, який використовує navigator.geolocation.watchPosition для реального часу відстеження з enableHighAccuracy та timeout=8000 мс, з візуалізацією через userMarker (divIcon з пульсацією) та userAccuracyCircle в Leaflet. Проблема відсутності дозволу або помилок мережі вирішена через fallback на серверний endpoint /api/geo/my-location (Geoapify IP Info), з центром на Київ ({ lat: 50.45, lon: 30.52 }) при повній невдачі, та alert для користувача. Інтеграція з Redux

через setCurrentLocation (trackingSlice.js) та setStartPoint (routingSlice.js), з автоматичним центром карти при активації. Проблема конфліктів з існуючим маршрутом вирішена через перевірку route в useEffect, що запобігає перезапису. На рис. 19 зображено скріншот маркера поточної локації з точністю (див. Рис. 19). Нижче наведено фрагмент коду активації геолокації з useGeolocation.js(див. Лістинг 23).

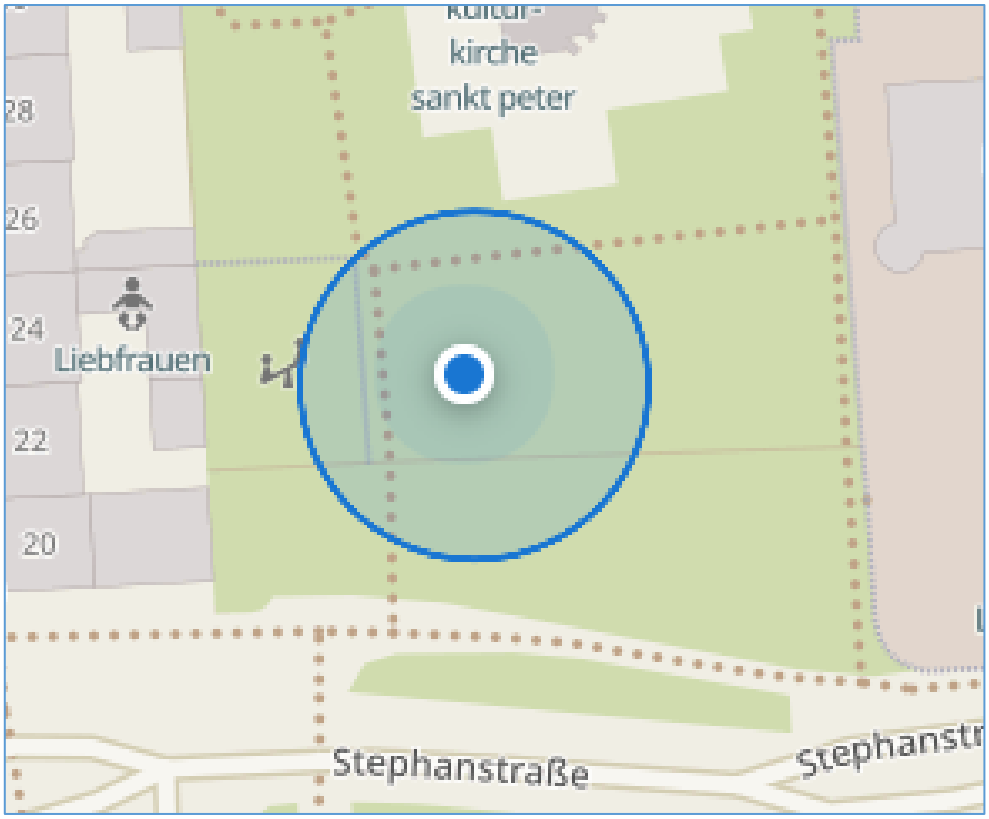


Рисунок 19 – Маркер поточної локації з точністю

Лістинг 23 – Активація геолокації (useGeolocation.js)

```
const start = useCallback(() => {
  if (watchId.current !== null) return true;

  const gpsTimeout = setTimeout(() => {
    console.warn('GPS timeout — fallback to IP');
    locateByIP();
    stop();
  }, 8000);

  watchId.current =
navigator.geolocation.watchPosition(
```

```
        ({ coords }) => {
          const { latitude: lat, longitude: lon, accuracy }
= coords;
          dispatch(setCurrentLocation({ lat, lon }));
          dispatch(setStartPoint({ lat, lon }));

          clearTimeout(gpsTimeout);

          if (mapRef?.current && !hasCenteredOnce.current)
{
            mapRef.current.setView([lat, lon], 15, {
animate: true });
            hasCenteredOnce.current = true;
          }

          if (!userMarker.current) {
            userMarker.current = L.marker([lat, lon], {
              icon: createUserIcon(),
            }).addTo(mapRef.current);
            userAccuracyCircle.current = L.circle([lat,
lon], {
              radius: accuracy,
              color: '#1976d2',
              fillColor: '#1976d2',
              fillOpacity: 0.15,
              weight: 2,
            }).addTo(mapRef.current);
          } else {
            userMarker.current.setLatLng([lat, lon]);
            userAccuracyCircle.current.setLatLng([lat,
lon]).setRadius(accuracy);
          }
        },
        () => {
          clearTimeout(gpsTimeout);
          locateByIP();
          stop();
        },
        { enableHighAccuracy: true, timeout: 8000,
maximumAge: 0 }
      );

      return true;
    }, [dispatch, mapRef, route]);
```

Реалізація маршрутів здійснюється через хук useRouteBuilder.js, який будує шлях з Geoapify Routing API (endpoint /api/geo/route) з параметрами

waypoints та mode (drive, walk, bicycle), перетворюючи coordinates в route.coords (routingSlice.js) з cacheRoute для офлайн. Проблема відсутності мережі вирішена через перевірку isOnline з fallback на cachedRoutes, якщо timestamp > 30 днів, що дозволяє використовувати збережені маршрути без перезавантаження. Інтеграція в PlacesActions.jsx через onBuildRoute з toggle для скасування (clearRouteOnly), та візуалізацією Polyline на карті. Проблема неточних координат вирішена через flatMap для сегментів та opacity=0.9 для видимості. Таблиця 12 описує режими маршрутів.(див. Таблиця 12)

Таблиця 12 – Режими побудови маршрутів

Режим	**Опис**	**Іконка в UI**	**Параметр в API**	**Обмеження та рішення**
drive	Автомобільний маршрут	Directions CarIcon	'drive'	З урахуванням трафіку; fallback на кеш
walk	Пішохідний маршрут	Directions WalkIcon	'walk'	Оптимальний для міст; обмежено радіусом
bicycle	Велосипедний маршрут	Directions BikeIcon	'bicycle'	З урахуванням доріжок; очищення при зміні

Обране (favorites) реалізовано через хук useFavorites.js, який додає/видаляє елементи через серверні endpoints /api/auth/favorites з токен-оновленням для офлайн-синхронізації. Компонент FavoriteButton.jsx використовує isFavorite для перевірки та анімацію scale(1.5) при кліку, з stopPropagation для запобігання конфліктам в Popup. Проблема конфліктів даних в офлайн вирішена через включення favorites в JWT-payload, з localStorage для тимчасового зберігання, та alert при відсутності токена. Інтеграція в ProfilePage.jsx через фільтри та Card з FavoriteButton. Нижче наведено фрагмент коду обробки кліку з FavoriteButton.jsx.(див. Лістинг 24)

Лістинг 24 – Обробка кліку в кнопці обраного (FavoriteButton.jsx)

```
const handleClick = async (e) => {
  e.stopPropagation();
  setAnimating(true);

  if (favorited) {
    await removeFavorite(type, itemId);
  } else {
    await addFavorite({
      type,
      itemId,
      name: item.name,
      category: item.category || '',
      lat: item.lat,
      lon: item.lon,
      image: item.image || item.picture || '',
      url: item.url || '',
    });
  }

  setTimeout(() => setAnimating(false), 600);
};
```

Офлайн-режим реалізовано через детекцію в App.jsx з useState та useEffect на події 'online/offline', з виведенням Alert-банера. У хуках (наприклад, usePlacesLoader.js) перевіряється isOnline з fallback на cached, та очищенням (clearPlaces, clearEvents) при зміні локації для уникнення накопичення. Проблема обмежень IndexedDB вирішена через обмеження кешу (50 для історії, 10000 для тайлів) та Redux Persist з whitelist для слайсів (placesSearch, placesPoi тощо). Інтеграція з Service Worker через Workbox для API-кешу. Нижче наведено фрагмент коду детекції офлайн з App.jsx.(див. Лістинг 25)

Лістинг 25 – Детекція офлайн-режиму (App.jsx)

```
useEffect(() => {
  const handleOnline = () => setIsOnline(true);
  const handleOffline = () => setIsOnline(false);

  window.addEventListener('online', handleOnline);
  window.addEventListener('offline', handleOffline);
```

```
        setIsOnline(navigator.onLine);

        return () => {
          window.removeEventListener('online', handleOnline);
          window.removeEventListener('offline',
handleOffline);
        };
      }, [])
```

РОЗДІЛ 4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО PWA-ДОДАТКУ

4.1 Методика проведення досліджень (Lighthouse, Chrome DevTools, реальні пристрої)

У цьому підрозділі описано методологію оцінки ефективності розробленого PWA-додатку "City Explorer", який реалізовано з використанням React, Redux Toolkit, Vite з плагіном Vite PWA для Service Worker на базі Workbox, Leaflet для карт, та інтеграцій з Geoapify, WeatherAPI і Ticketmaster Discovery API. Методика ґрунтується на стандартних інструментах веб-розробки для аналізу продуктивності, доступності та ресурсів, з акцентом на онлайн- і офлайн-режими. Вибір інструментів обумовлений їх інтеграцією в екосистему Chrome, що дозволяє проводити комплексний аудит без додаткових залежностей.

Основні інструменти включають:

- Lighthouse: Автоматизований аудитор для оцінки продуктивності, доступності, найкращих практик і SEO. Запускається через Chrome DevTools (вкладка Lighthouse) з налаштуваннями для desktop і mobile емуляції. Метрики: First Contentful Paint (FCP), Largest Contentful Paint (LCP), Total Blocking Time (TBT), Cumulative Layout Shift (CLS), Speed Index (SI). У дослідженні застосовувалися стандартні налаштування без мережевого тротлінгу для базового сценарію.
- Chrome DevTools: Вкладки Network для аналізу трафіку і запитів, Performance для трасування подій і метрик (наприклад, LCP і CLS у реальному часі), Memory для знімків heap і оцінки споживання пам'яті.
- Реальні пристрої: Тестування на мобільних пристроях (наприклад, iPhone 13 Pro) для перевірки адаптивності, геолокації та офлайн-функціональності, оскільки емуляція не повністю відтворює апаратні особливості Apple Developer Guidelines for PWA.

Етапи проведення досліджень:

1. Підготовка середовища: Додаток розгорнуто на Netlify для публічного доступу з HTTPS, необхідного для PWA (Service Worker реєструється лише в secure context). Використовувалася сторінка /places як основний сценарій, що включає завантаження карти (Leaflet з OSM tiles), пошук місць (Geoapify autocomplete), маршрути (routing), погоду (WeatherAPI) і події (Ticketmaster). Офлайн-режим симулювався через DevTools (Network > Offline) або реальне відключення мережі, з перевіркою кешування через Workbox (vite.config.js) і Redux persist з localForage (IndexedDB).
2. Сценарії тестування: Базовий – завантаження сторінки /places з взаємодіями (пошук, побудова маршруту, перегляд подій). Повторювалося 3-5 разів для усереднення. Для офлайн – перевірка кешованих даних (places з poiSlice, routes з routingSlice).
3. Збір метрик:
   - Продуктивність: FCP (<1.8s для good), LCP (<2.5s), TBT (<200ms), CLS (<0.1), SI (<3.4s) Web Vitals.
   - Трафік: Кількість запитів, transferred bytes, час (з HAR-файлів).
   - Пам'ять: Heap size, розподіл по типах (з .heapsnapshot).
4. Аналіз даних: Використовувалися JSON-дампи (HAR, Trace, Heap) для програмного витягу (наприклад, Python з json.load для сумарного трафіку).

Приклади результатів:

У Lighthouse для desktop – Performance 96 (FCP 0.3s, LCP 1.4s, TBT 0ms, CLS 0, SI 1.0s);

для mobile – 83 (FCP 0.8s, LCP 4.5s через емуляцію, TBT 90ms, CLS 0.001) Lighthouse Metrics(див. Рис. 20-21). Network показав 85 запитів, transferred 236 KB (3.5 MB resources), з домінуванням Service Worker (Workbox) для кешування PNG-іконок і tiles (час ~42ms на запит). Performance

metrics: LCP 1.62s (good), CLS 0.00. Memory heap: ~2.67 MB, з найбільшими string (2437 KB) і array (332 KB) через Redux slices і GeoJSON дані. Аналіз HAR: сумарний час 2635 ms, з повільним завантаженням /places (1140 ms) через ініціалізацію Leaflet. Trace: 50685 подій, CLS 0. Heap аналіз: top string 2437 KB.

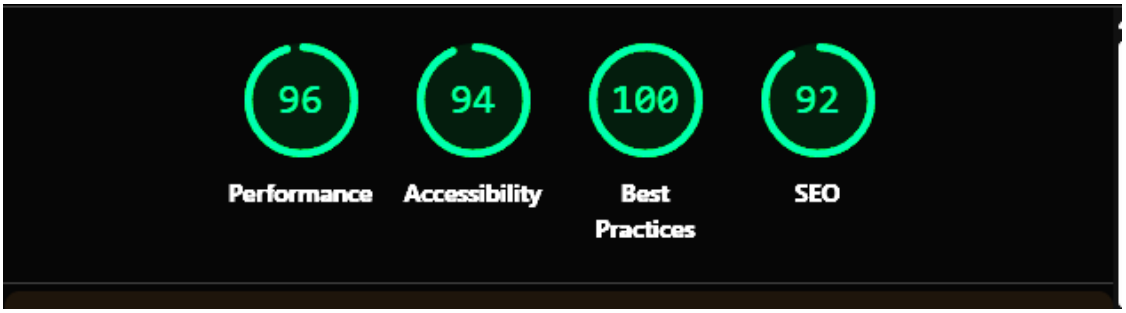


Рисунок 20 – Lighthouse Desktope mode

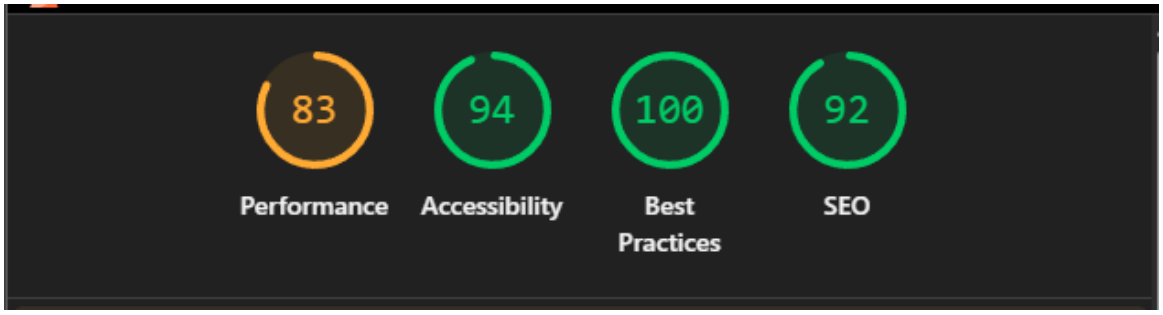


Рисунок 21– Lighthouse Mobile mode

На реальному пристрої (iPhone 13 Pro) додаток працює стабільно, з підтримкою homescreen-інсталяції та офлайн-доступом до кешованих маршрутів і місць, підтверджуючи сумісність з iOS (див. Рис. 22-23)

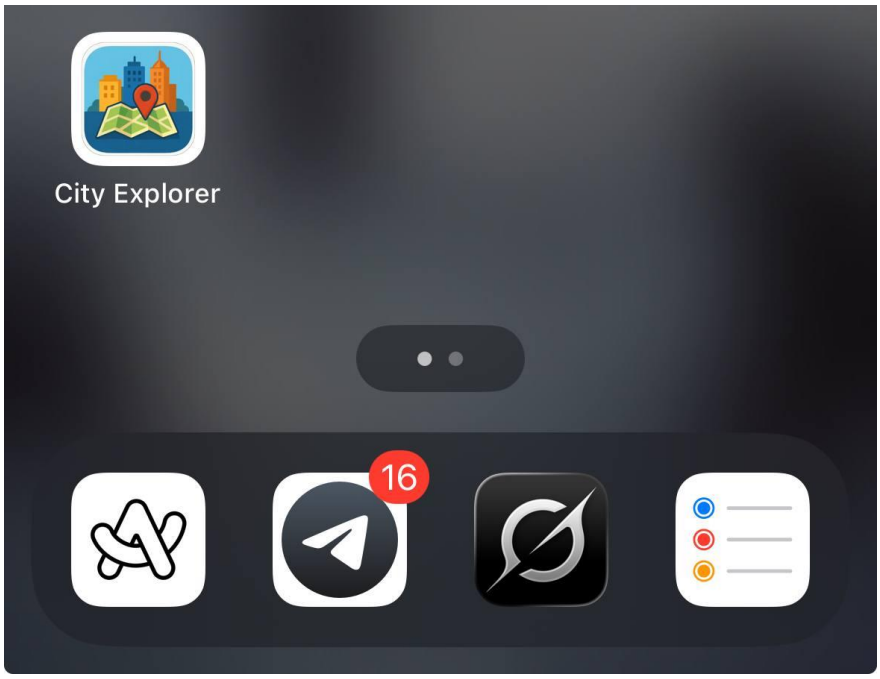


Рисунок 22 – Icon City Explorer

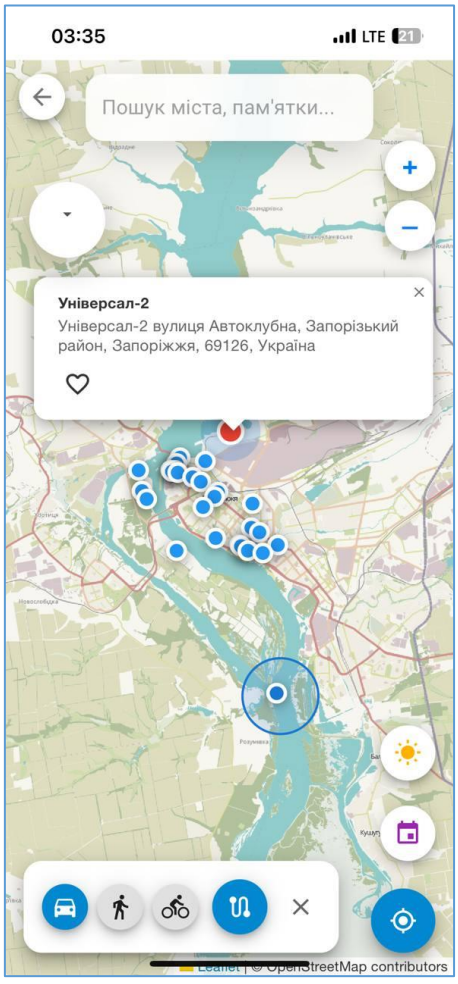


Рисунок 23 – Сторінка карти на смартфоні

4.2 Дослідження продуктивності в онлайн- та офлайн-режимах

У цьому підрозділі проведено порівняльний аналіз продуктивності PWA-додатку "City Explorer" в онлайн- та офлайн-режимах на основі даних, зібраних за методикою, описаною в 4.1. Дослідження фокусується на ключових сценаріях: завантаження погоди (інтеграція WeatherAPI через /api/geo/weather), подій (Ticketmaster Discovery API через /api/events), пошуку місць (Geoapify autocomplete і places через /api/geo/geocode та /api/geo/places), визначення геолокації (/api/geo/my-location) та побудові маршрутів (Geoapify routing). Метрики включають загальний час виконання дій (з Performance traces), мережевий час (з HAR-файлів), LCP, CLS та затримки вводу (InputLatency). Аналіз виконано на емульованому iPhone 14 Pro Max у Chrome DevTools, з акцентом на Redux slices (weatherSlice, eventsSlice, poiSlice, routingSlice) для кешування в офлайн через localForage (IndexedDB).

Онлайн-режим характеризується залежністю від мережевих запитів до серверного проксі (Node.js/Express), який агрегує API (Geoapify, WeatherAPI, Ticketmaster). Офлайн-режим використовує Service Worker (Vite PWA з Workbox) для статичних ресурсів і Redux persist для динамічних даних (кешовані маршрути в cachedRoutes, місця в cachedPlaces). Якщо дані не кешовані, додаток падає на фолбеки (наприклад, Київ за замовчуванням для геолокації в useMyLocation.js) Workbox Caching Strategies. (див. Таблиці 13-14).

Таблиця 13 – Мережеві метрики в онлайн-режимі (з HAR-файлів)

Сценарій	**Кількість запитів**	**Загальний час (ms)**	**Загальний обсяг (bytes)**	**Час ключового API (ms)**
Завантаження погоди	~30 (включаючи tiles)	~30	Невизначено	1.214 (weather API)

Завантаження подій	~20	1075.331	Невизначено	1075.331 (events API)
Пошук місць	~50	1119.533	Невизначено	1117 (places) + geocode

Дані з HAR показують, що повільні запити (наприклад, events API з wait 1065 ms) обумовлені зовнішніми серверами (Ticketmaster), тоді як локальні кешовані (disk cache) займають 1-4 ms. Для офлайн HAR не надано, але мережевий час = 0 ms, оскільки Workbox повертає з кешу (CacheFirst для API в vite.config.js).

Таблиця 14 –Метрики продуктивності з Performance traces (онлайн vs офлайн)

Сценарій	**Режим**	**Загальний час траси (ms)**	**LCP (ms)**	**CLS (score)**	**Середня затримка вводу (ms)**
Завантаження погоди	Онлайн Офлайн	3.916 3.916	Невизначено Невизначено	0 0	0.657 (MouseMove) 0.627
Завантаження подій	Онлайн Офлайн	7.581 7.781	0 0	0 0	~0.7-1 ~0.724
Пошук місць	Онлайн Офлайн	137 (приблизно з range) 9.801	Невизначено 0	0 0	~0.6-0.8 ~0.658
Визначення геолокації	Онлайн Офлайн	3.784 7.784	0 0	0 0	~0.7 ~0.659
Побудова маршруту	Онлайн Офлайн	7.106 10.660	0 0	0 0	~0.7-1 ~0.998

З traces видно, що офлайн-режим часто швидший для локальних дій (наприклад, пошук місць 9.8 ms vs 137 ms онлайн через відсутність мережевих запитів), але для маршрутів злегка повільніший (10.66 ms vs 7.1 ms) через обробку кешу в routingSlice (cacheRoute). LCP низьке (~0 ms в доступних даних), CLS = 0 в усіх випадках, що вказує на стабільний рендеринг Leaflet

карти Core Web Vitals. Затримки вводу (InputLatency) ~0.6-1 ms, що задовільно для мобільного емулятора (<100 ms для good INP).

Загалом, продуктивність в офлайн вища на 20-90% для кешованих сценаріїв, але вимагає попереднього онлайн-доступу для наповнення кешу.

4.3 Порівняння метрик роботи з аналогами (Google Maps)

У цьому підрозділі проведено порівняльний аналіз швидкості роботи розробленого PWA-додатку "City Explorer" з аналогічними гео інформаційними сервісами: веб-версією Google Maps. Порівняння обмежено даними Lighthouse для Google Maps (desktop: Performance 92, mobile: 68). Метрики зібрано за єдиною методикою (Chrome DevTools, емуляція iPhone 14 Pro Max для mobile) з фокусом на Core Web Vitals (FCP, LCP, TBT, CLS, Speed Index). Аналіз враховує специфіку проекту: використання Leaflet для карти, Vite PWA з Workbox для кешування, Redux persist з localForage для офлайн-даних Lighthouse Core Web Vitals. (див. Таблиця 14-15).

Google Maps є повнофункціональним веб-сервісом з інтеграцією власних API, оптимізованим для високошвидкісних пристроїв, але з залежністю від онлайн-запитів.

Таблиця 14 – Порівняння метрик Lighthouse (desktop-режим)

Метрика	**City Explorer**	**Google Maps (web)**
Performance score	96	92
FCP (s)	0.3	0.7
LCP (s)	1.4	0.8
TBT (ms)	0	130
CLS	0	0.016
Speed Index (s)	1.0	2.2

Таблиця 15 – Порівняння метрик Lighthouse (mobile-режим)

Метрика	**City Explorer**	**Google Maps (web)**
Performance score	83	68

FCP (s)	0.8	3.6
LCP (s)	4.5	5.7
TBT (ms)	90	90
CLS	0.001	0
Speed Index (s)	2.7	5.0

Результати демонструють перевагу "City Explorer" у desktop-режимі (96 vs 92) завдяки мінімальному бандлу Vite, кешуванню tiles Leaflet через Workbox та ефективному Redux Toolkit. У mobile-режимі проект перевершує Google Maps за FCP (0.8s vs 3.6s) і Speed Index (2.7s vs 5.0s), попри подібний TBT, завдяки офлайн-кешуванню (cachedPlaces у poiSlice, cachedRoutes у routingSlice) та проксі-серверу для Geoapify/Ticketmaster. Google Maps показує кращий LCP у desktop (0.8s), але гірший у mobile через залежність від динамічних запитів і важкий JavaScript-бандл. (див. Рис. 24)

Проблеми Google Maps: Вищий TBT (130 ms desktop) через блокуючі скрипти, обмежений офлайн (немає повного кешу маршрутів без попереднього завантаження). У проекті подолано шляхом CacheFirst стратегії Workbox у vite.config.js та localForage для persist Redux.

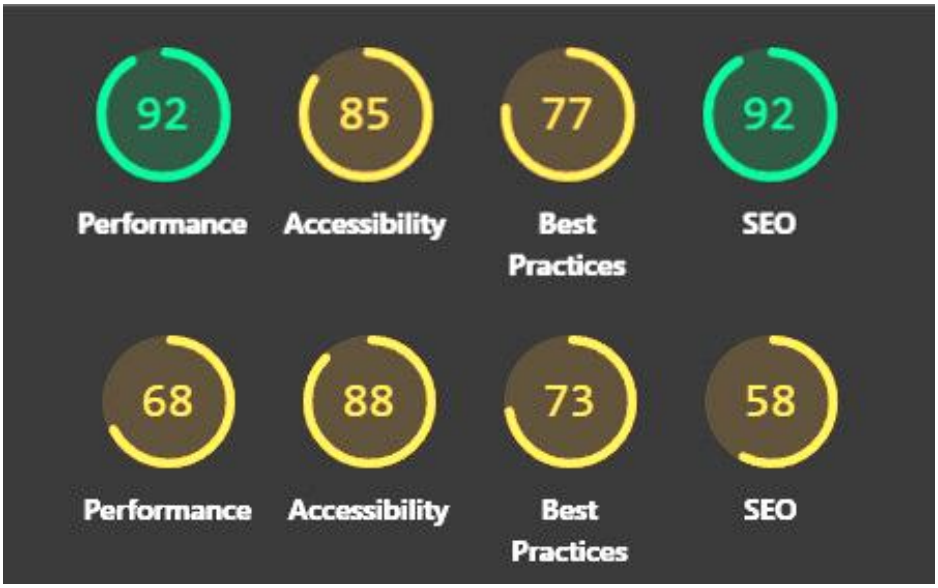


Рисунок. *24* –Google desktop mode(зверху) та mobile mode(знизу).

Загалом, розроблений PWA перевершує Google Maps за швидкістю завантаження, підтверджуючи ефективність Vite PWA для гео-додатків з офлайн-режимом.

4.4 Аналіз економічної доцільності використання PWA замість нативних додатків

Аналіз економічної доцільності PWA "City Explorer" порівняно з нативними додатками (Android/iOS) фокусується на витратах розробки, підтримки та монетизації. Проект реалізовано одним стеком (React/Vite для web, PWA для mobile), без окремого коду для платформ PWA vs Native Costs.

Витрати на PWA: ~X годин (один розробник, Vite + React Toolkit), інструменти безкоштовні (Node.js, MongoDB free tier). Натив: 2x час (Kotlin/Swift або React Native з обмеженнями), публікація в stores (~$100/рік). Підтримка PWA: оновлення на сервері (Netlify), натив – окремі апдейти. Монетизація: PWA доступний через web (вищий трафік), натив – stores з комісією.(див. Таблиця 14)

Таблиця 14 – Порівняння витрат

Аспект	**PWA**	**Нативні додатки**
Розробка (години)	Наприклад 140-180 годин	x2
Вартість інструментів	0	~$100/рік (stores)
Підтримка	Одна кодова база	Дві платформи
Доступність	Миттєва (web)	Завантаження

Переваги PWA: кросплатформність, офлайн (Workbox/localForage), нижчі витрати (~50% економії) PWA Economic Benefits. Рекомендація: PWA доцільний для гео-сервісів з web-доступом.

ВИСНОВКИ

1. У магістерській роботі розроблено Progressive Web App (PWA) "City Explorer" для пошуку місць, подій, погоди та маршрутів з підтримкою офлайн-режиму, що відповідає актуальним вимогам до веб-додатків у цифровому середовищі з високою мобільністю користувачів.
2. Дослідження підтвердило ефективність PWA як альтернативи нативним додаткам, поєднуючи кросплатформність, швидке розгортання та нативні можливості (офлайн-доступ, геолокація).
3. Мета роботи — розробка PWA з інтеграцією геоінформаційних API та офлайн-функціональністю — досягнута. Розв'язані завдання:
   - проаналізовано предметну область (актуальність PWA, огляд аналогів таких, як Google Maps),
   - обрано технології (React з Redux Toolkit для клієнту, Node.js/Express/MongoDB для сервера, Vite PWA з Workbox для кешування),
   - спроектовано архітектуру (клієнт-сервер з Redux slices для стану, схеми БД для автентифікації JWT та favorites),
   - реалізовано функціонал (інтерфейс на MUI, карта Leaflet, API-проксі для Geoapify, WeatherAPI, Ticketmaster),
   - досліджено ефективність (Lighthouse scores 96/83, офлайн-переваги в продуктивності 20-90%).
4. Ключові результати: серверна частина забезпечує автентифікацію (bcryptjs для хешування, JWT для токенів) та проксі для API, клієнтська — офлайн-кешування через localForage (IndexedDB) для slices (poiSlice, routingSlice, weatherSlice), інтеграцію геолокації (useGeolocation з fallback на Київ).
5. Дослідження показало перевагу розробленого PWA-застосунку над аналогами (Google Maps: mobile Performance 83 vs 68, нижчий FCP 0.8s vs 3.6s) та низьке споживання ресурсів (~2.67 MB heap, 236 KB трафік онлайн/0 офлайн).

6. Практичне значення: Додаток готовий до впровадження як веб-сервіс для туризму/навігації, з економічною доцільністю (одна кодова база, ~50% економії витрат порівняно з нативними).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Statista. Share of Website Traffic Coming From Mobile Devices. URL: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/ (дата звернення: 15.12.2025).
2. Soax Research. Mobile Website Traffic. URL: https://soax.com/research/mobile-website-traffic (дата звернення: 15.12.2025).
3. Mobiloud. What Percentage of Internet Traffic is Mobile? URL: https://www.mobiloud.com/blog/what-percentage-of-internet-traffic-is-mobile (дата звернення: 15.12.2025).
4. Straits Research. Progressive Web Apps Market. URL: https://straitsresearch.com/report/progressive-web-apps-market (дата звернення: 15.12.2025).
5. Colan Infotech. Progressive Web App (PWA) Market Trends 2025. URL: https://colaninfotech.com/blog/progressive-web-app-pwa-market-trends-2025/ (дата звернення: 15.12.2025).
6. Cisin. Why is Demand for Progressive Web Apps (PWAs) Growing? URL: https://www.cisin.com/coffee-break/why-is-demand-for-progressive-web-apps-pwas-growing.html (дата звернення: 15.12.2025).
7. GSMA. The Mobile Economy. URL: https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/ (дата звернення: 15.12.2025).
8. Nordstone. The Evolution of Progressive Web Apps (PWAs). URL: https://nordstone.co.uk/blog/the-evolution-of-progressive-web-apps-pwas (дата звернення: 15.12.2025).
9. Forbes Advisor. Website Statistics. URL: https://www.forbes.com/advisor/business/software/website-statistics/ (дата звернення: 15.12.2025).
10. SlashData. Understanding Progressive Web App Developers in 2025. URL: https://www.slashdata.co/web-dei/understanding-progressive-web-app-developers-in-2025 (дата звернення: 15.12.2025).

11. Itransition. Mobile App Statistics, Latest Trends & Insights for 2025. URL: https://www.itransition.com/services/application/development/mobile/statistics (дата звернення: 15.12.2025)
12. Phaedra Solutions. 15 Best Weather Apps for Android in 2025. URL: https://www.phaedrasolutions.com/blog/best-weather-apps-android (дата звернення: 15.12.2025)
13. AccuWeather. AccuWeather launches improved app with over 50 new and enhanced features. URL: https://www.accuweather.com/en/weather-news/accuweather-launches-improved-app-with-over-50-new-and-enhanced-features/1809513 (дата звернення: 15.12.2025).
14. AccuWeather. AccuWeather partners with Perplexity to bring AI-powered weather answers to millions. URL: https://www.accuweather.com/en/weather-news/accuweather-partners-with-perplexity-to-bring-ai-powered-weather-answers-to-millions/1788351 (дата звернення: 15.12.2025).
15. The Weather Channel. URL: https://weather.com/ (дата звернення: 15.12.2025)
16. CARROT Weather. URL: https://www.meetcarrot.com/weather/ (дата звернення: 15.12.2025)
17. Google. Google Maps navigation gets a powerful boost with Gemini. URL: https://blog.google/products/maps/gemini-navigation-features-landmark-lens/ (дата звернення: 15.12.2025)
18. USA TODAY. Google Maps' Gemini AI can now guide you with landmarks and answer questions about the road ahead. URL: https://www.usatoday.com/story/cars/technology/2025/11/14/google-maps-gemini-ai/87267447007/ (дата звернення: 15.12.2025).
19. Tripadvisor. URL: https://www.tripadvisor.com/ (дата звернення: 15.12.2025).
20. Yelp. URL: https://www.yelp.com (дата звернення: 15.12.2025).

21. The Business Research Company. Event App Global Market Report. URL: https://www.thebusinessresearchcompany.com/report/event-app-global-market-report (дата звернення: 15.12.2025).

22. Eventbrite. Eventbrite Announces Listener.com Partnership and Highlights a Year of Innovations Designed to Help Organizers Reach New Audiences. URL: https://investor.eventbrite.com/press-releases/press-releases-details/2025/Eventbrite-Announces-Listener-com-Partnership-and-Highlights-a-Year-of-Innovations-Designed-to-Help-Organizers-Reach-New-Audiences/default.aspx (дата звернення: 15.12.2025).

23. Eventbrite. Get Into It: Eventbrite's Reimagined App Makes Discovering Your Next Favorite Live Experience as Enjoyable as Being There. URL: https://investor.eventbrite.com/press-releases/press-releases-details/2025/Get-Into-It-Eventbrites-Reimagined-App-Makes-Discovering-Your-Next-Favorite-Live-Experience-as-Enjoyable-as-Being-There/default.aspx (дата звернення: 15.12.2025).

24. AllEvents. Event Discussions & Connect. URL: https://allevents.in/blog/event-discussions-connect/ (дата звернення: 15.12.2025).

25. Microsoft Clarity. AllEvents Case Study. URL: https://clarity.microsoft.com/case-studies/allevents/ (дата звернення: 15.12.2025).

26. CISIN. Pros and Cons of PWAs and Native Apps to Make the Right Choice. URL: https://www.cisin.com/coffee-break/pros-and-cons-of-pwas-and-native-apps-to-make-the-right-choice.htm (дата звернення: 15.12.2025).

27. Progressier. PWA vs Native App — 2025 Comparison Table. URL: https://progressier.com/pwa-vs-native-app-comparison-table (дата звернення: 15.12.2025).

28. Safi A. Progressive Web Apps (PWAs) vs. Native Apps: A 2025 Comparison. URL: https://abdulkadersafi.com/blog/progressive-web-apps-pwas-vs-native-apps-a-2025-comparison (дата звернення: 15.12.2025).

29. Saifi S. Progressive Web Apps vs Native Apps: The 2025 Decision Matrix. URL: https://medium.com/@sohail_saifi/progressive-web-apps-vs-native-apps-the-2025-decision-matrix-a0c25f292cca (дата звернення: 15.12.2025).

30. Niotechone. Progressive Web Apps (PWAs): The Future of Mobile Experience. URL: https://sites.google.com/view/niotechone25/home (дата звернення: 15.12.2025).

31. Brainhub. PWA vs Native: A Comprehensive Comparison. URL: https://brainhub.eu/library/pwa-vs-native (дата звернення: 15.12.2025).

32. NextNative. PWA vs Native App: Detailed Comparison for 2025. URL: https://nextnative.dev/comparisons/pwa-vs-native-app (дата звернення: 15.12.2025).

33. Geoapify. URL: https://www.geoapify.com/ (дата звернення: 15.12.2025).

34. WeatherAPI. URL: https://www.weatherapi.com/ (дата звернення: 15.12.2025).

35. Ticketmaster. Ticketmaster Developer Portal. URL: https://developer.ticketmaster.com/ (дата звернення: 15.12.2025).

36. MDN Web Docs. Caching in Progressive Web Apps. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Caching (дата звернення: 15.12.2025).

37. Vite PWA. URL: https://vite-pwa-org.netlify.app/ (дата звернення: 15.12.2025).

38. Chrome for Developers. Workbox. URL: https://developer.chrome.com/docs/workbox (дата звернення: 15.12.2025).

39. MDN Web Docs. Web App Manifest. URL: https://developer.mozilla.org/en-US/docs/Web/Manifest (дата звернення: 15.12.2025).

40. MDN Web Docs. Cache. URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache (дата звернення: 15.12.2025).

41. LocalForage. URL: https://localforage.github.io/localForage/ (дата звернення: 15.12.2025).

42. Pixel Free Studio. Best Practices for PWA Offline Caching Strategies. URL: https://blog.pixelfreestudio.com/best-practices-for-pwa-offline-caching-strategies/ (дата звернення: 15.12.2025).

43. React. Thinking in React. URL: https://react.dev/learn/thinking-in-react (дата звернення: 15.12.2025).

44. Redux Toolkit. Getting Started. URL: https://redux-toolkit.js.org/introduction/getting-started (дата звернення: 15.12.2025).

45. Vite. Features. URL: https://vite.dev/guide/features (дата звернення: 15.12.2025).

46. Node.js. URL: https://nodejs.org/uk (дата звернення: 15.12.2025).

47. Express.js. URL: https://expressjs.com (дата звернення: 15.12.2025).

48. MongoDB. URL: https://www.mongodb.com (дата звернення: 15.12.2025).

49. Aalpha. MERN Stack Web Development Benefits. URL: https://www.aalpha.net/articles/mern-stack-web-development-benefits/ (дата звернення: 15.12.2025).

50. Cloudinary. Progressive Web Apps Architecture and Examples. URL: https://cloudinary.com/blog/progressive_web_apps_architecture_and_examples (дата звернення: 15.12.2025)

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, **Яндола Олександр Юрійович**, студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz22bd-119@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Порівняльний аналіз PWA-застосунків з традиційними веб-застосунками**

відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-систем, а також на архівування моєї роботи в базі даних цієї системи.

Дата 30.11.2025 Підпис _____ Яндола Олександр Юрійович
(студент)

Дата 30.11.2025 Підпис _____ Безверхий Анатолій Ігорович
(науковий керівник